



TAMPERE UNIVERSITY OF TECHNOLOGY

MICHELE VALENTI
CONVOLUTIONAL NEURAL NETWORKS FOR ACOUSTIC
SCENE CLASSIFICATION

Master of Science Thesis

Examiner: Tuomas Virtanen, Stefano
Squartini, Aleksandr Diment
Examiner and topic approved by the
Faculty Council of Computing and
Electrical Engineering
on 4 May 2016

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY

Master's Degree Programme in Signal Processing

VALENTI, MICHELE: Convolutional neural networks for acoustic scene classification

Master of Science Thesis, 49 pages

September 2016

Major: Signal Processing

Examiners: Prof. Tuomas Virtanen, Prof. Stefano Squartini, Diment Aleksandr

Keywords: Acoustic scene classification, convolutional neural networks, DCASE, computational audio processing

In this thesis we investigate the use of deep neural networks applied to the field of computational audio scene analysis, in particular to acoustic scene classification. This task concerns the recognition of an acoustic scene, like a park or a home, performed by an artificial system. In our work we examine the use of deep models aiming to give a contribution in one of their use cases which is, in our opinion, one of the most poorly explored.

The neural architecture we propose in this work is a convolutional neural network specifically designed to work on a time-frequency audio representation known as log-mel spectrogram. The network output is an array of prediction scores, each of which is associated with one class of a set of 15 predefined classes. In addition, the architecture features batch normalization, a recently proposed regularization technique used to enhance the network performance and to speed up its training.

We also investigate the use of different audio sequence lengths as classification unit for our network. Thanks to these experiments we observe that, for our artificial system, the recognition of long sequences is not easier than of medium-length sequences, hence highlighting a counterintuitive behaviour. Moreover, we introduce a training procedure which aims to make the best of small datasets by using all the labeled data available for the network training. This procedure, possible under particular circumstances, constitutes a trade-off between an accurate training stop and an increased data representation available to the network. Finally, we compare our model to other systems, proving that its recognition ability can outperform either other neural architectures as well as other state-of-the-art statistical classifiers, like support vector machines and Gaussian mixture models.

The proposed system reaches good accuracy scores on two different databases collected in 2013 and 2016. The best accuracy scores, obtained according to two cross-validation setups, are 77% and 79% respectively. These scores constitute a 22% and 6.1% accuracy increment with respect to the correspondent baselines published together with datasets.

PREFACE

The work reported in this thesis has been done during an internship at the Department of Signal Processing at Tampere University of Technology between March and August 2016.

To begin, I wish to express my deepest gratitude to Professors Tuomas Virtanen and Stefano Squartini for giving me the chance to live a fulfilling work and life experience. I especially thank Prof. Virtanen for his encouraging kindness in welcoming and supporting me throughout the whole internship.

A special thanks also goes to Giambattista Parascandolo and Aleksandr Diment. I consider their tireless supervision and support, along with their significant professional advices, to have been crucial for the accomplishment of this work. In addition, I wish to extend my thanks to the Audio Research Group of the Tampere University of Technology for creating a stimulating and positive working environment through its active and enthusiastic attitude. In particular, I wish to thank Annamaria Mesaros, for her constant help and advice, and Toni Heittola, for providing the base of the code used for this work. I also wish to acknowledge CSC — IT Center for Science, Finland, for generous computational resources.

This thesis concludes a two-years path marked by the birth of countless and priceless friendships. I would like to thank all my university mates at Università Politecnica delle Marche for the selfless and precious support during courses and exam preparation. In addition, I thank all my Italian and Finnish friends for their constructive presence and joyful support. Without them, this path would have been far from worth fulfilling.

Finally, the most special thanks goes to my family. To my brother, sister, parents and grandparents, who have never failed to believe in me and support me to the very best of their ability.

Michele Valenti
29 September 2016

CONTENTS

1. Introduction	1
2. Background	4
2.1 Supervised learning	4
2.2 Neural networks	8
2.3 Audio features	22
2.4 Previous work	24
3. Methodology	27
3.1 Feature extraction and pre-processing	27
3.2 Proposed neural network	29
3.3 Training and regularization	30
4. Evaluation	35
4.1 Dataset and metrics	35
4.2 Baseline system	37
4.3 Network parameter experiments	38
4.4 Main experiments and results	40
4.5 DCASE evaluations	44
4.6 Discussion	46
5. Conclusions	48
References	49

TERMS AND DEFINITIONS

ML	Machine learning
NN	Neural network
CNN	Convolutional neural network
ASC	Acoustic scene classification
CASA	Computational auditory scene analysis
DCASE	Detection and classification of acoustic scenes and events
SL	Supervised learning
GMM	Gaussian mixture model
GD	Gradient descent
ReLU	Rectified linear unit
FNN	Feed-forward neural network
MLP	Multilayer perceptron
RF	Receptive field
BP	Backpropagation
SGD	Stochastic gradient descent
DFT	Discrete Fourier transform
STFT	Short-time Fourier transform
RNN	Recurrent neural network
HMM	Hidden Markov model
LSP	Line spectral frequency
MFCC	Mel-frequency cepstral coefficient
SVM	Support vector machine
DCT	Discrete cosine transform
EM	Expectation-maximization

GPU	Graphics processing unit
NMF	Non-negative matrix factorization

1. INTRODUCTION

Artificial intelligence is a discipline that has been intriguing the human mind even before its very name was coined. In modern science, artificial intelligence is the field that studies how to make a machine able to do things that require the human intelligence to be done. Nowadays, machines are able to efficiently handle and manipulate a wide variety of multimedia content, but can they actually observe or listen to it? Are they really able to *understand* the data they are storing or reproducing? Evidently, these questions have all the same negative answer.

Machine learning (ML) is a computer science field that has its roots in the late fifties, when the merging of computer science and artificial intelligence started to raise interest throughout the scientific community. Arthur Lee Samuel, developer of one of the first self-learning algorithms, defined ML as the “field of study that gives computers the ability to learn without being explicitly programmed” [1]. This work focuses on a particular branch of ML, a branch born in the early sixties but in which interest has been exponentially growing mostly in the recent years, thanks to the computational power now at our disposal. This field is known as “deep learning” and it is defined as the study of algorithms and software structures built with the objective of extracting high-level abstractions from data. When talking of deep learning, it is common to refer to a family of non-linear learning architectures inspired by our very brain’s structure, i.e. neural networks (NNs).

Nowadays, NNs reached outstanding results in different fields of computational science, like computer vision and machine hearing [2]. Thanks to the definition of an ad-hoc training algorithm, NNs are able to autonomously learn underlying structures in raw or slightly pre-processed data, therefore sparing programmers and engineers from handcrafting high-level data properties. As deep learning research proceeds, state-of-the-art artificial systems (like the GoogLeNet [3] for image recognition) are becoming more and more able to handle very complex problems, now making us question the very concept of “creativity” [4, 5, 6].

In this thesis we study the use of a family of NNs — i.e. convolutional neural networks (CNNs) — applied to the branch of machine hearing known as “acoustic scene classification” (ASC). Falling under the umbrella of “computational auditory scene analysis” (CASA), ASC can be defined as the goal of classifying a recording into one of predefined classes, namely the environment in which the recording was

done. In other words, deep learning for ASC deals with the artificial analysis of a mixture of sounds from which the system autonomously learns to understand high-level data representations. Then, based on pre-learned feature-class associations, the system classifies each mixture choosing one class from the given set.

Our interest in artificial ASC lays in its many possible applications. For example, in [7] the authors explain how a well-designed context-aware device is supposed to have knowledge of a wide variety of information — light and noise level, communication costs and bandwidth — but not least acoustic scene information. By exploiting this information it is possible to create or enhance applications capable to react in different ways depending on the surrounding environment or its change. In [8] ML algorithms for context awareness are used for the development of intelligent wearable interfaces, whereas in [9] ASC is used to enhance the visual information in mobile robot navigation, thus allowing a high-level environment characterization.

NNs have been successfully used in many audio-related tasks, some examples being polyphonic sound event detection [10, 11], speech recognition [12] and note transcription [13]. Here we approach ASC by studying the use of a CNN-based classifier built and tested within the framework of the “Detection and Classification of Acoustic Scenes and Events” (DCASE) challenge of the Institute of Electrical and Electronics Engineers Audio and Acoustic Signal Processing Technical Committee (IEEE AASP TC) in 2016. This challenge is conceived to promote research in four different fields: ASC, sound event detection in synthetic and real life audio, and domestic audio tagging. More details about the DCASE challenges held in 2013 and 2016 will be given at the end of Chapter 2 and in Chapter 4 respectively.

Throughout the years, and especially during the DCASE challenges, many different models have been proposed for the task of ASC. Despite this, the use of CNNs has remained unexplored until the 2016 challenge, when systems finally started featuring these architectures either as stand-alone models or combined with others. Therefore, this work aims to present a novel contribution to this task showing how a CNN can be designed and trained in order to reach a significantly high performance.

Despite the strength of NNs has been theoretically and practically proved, they are not exempted from downsides. The need for a large amount of data, for example, is what mainly limits these powerful tools from making the best of their “representation capacity”, therefore not letting them extract optimal or useful information from the data. Because of this, in this work we propose a training method aimed to optimize the use of a restricted dataset in order to exploit all available information under specific circumstances. Moreover, we show how the use of a recently-proposed technique for training acceleration can be effectively introduced with additional benefit for the overall system performance.

The structure of this thesis is the following. In Chapter 2 we describe those ML

concepts and algorithms needed for a proper comprehension of NNs. After this, we focus on a description of the convolutional architecture, therefore explaining the theoretical characteristics that lead us to choose a CNN as the neural model proposed in this work. Finally, an overview of the most relevant ASC works is given in this chapter. In Chapter 3 we describe in detail the proposed architecture along with the training regularization and optimization techniques used in our experiments. In Chapter 4 we report a series of tests that are divided into two phases. In the first phase we aim to find the optimal set of parameters to be used in our network, so we will explore some differences between computational times and accuracy scores observed with different parameters and regularization methods. Then, in the second phase we evaluate the chosen architecture with different input characteristics and we compare it to other neural and non-neural classifiers. Finally, a discussion about the proposed solution, some of the problems encountered and possible future research areas is reported in Chapter 5.

2. BACKGROUND

In this chapter we introduce a background overview of the topics treated in this thesis. Here, the concepts of supervised learning (SL) and NNs are explained, along with a mathematical description of the NNs' training algorithm. Finally, after a description of the audio feature extraction process, we will briefly overview some of the main previous works conducted in the ASC field.

2.1 Supervised learning

SL is that ML task that can be formally defined as the task of inferring a function from labeled training data. In other words, similarly to how students learn at school with the help of a teacher, in a SL algorithm the machine will augment its “knowledge” by being subjected to examples of correct input-output associations, which we will call *training set* hereafter. The training set is here formally presented as an ensemble of array pairs: $(\mathbf{x}^{(o)}, \mathbf{y}^{(o)})$ with $o = 1, \dots, O$, where O is the number of training samples. We will refer to $\mathbf{x}^{(o)}$ as *feature vector* and to $\mathbf{y}^{(o)}$ as its corresponding *target vector*. Following the introduced notation, the final task of a SL algorithm is to make the model able to reproduce a function f which correctly associates each feature vector to its target vector:

$$\mathbf{y} = f(\boldsymbol{\theta}, \mathbf{x}) \text{ with } \mathbf{x} \in \mathbf{X}, \mathbf{y} \in \mathbf{Y}. \quad (2.1)$$

In Eq. (2.1) \mathbf{X} and \mathbf{Y} are called *feature space* and *target space* respectively, whereas $\boldsymbol{\theta}$ is the model's optimal set of parameters. The feature space is most generally a multidimensional dense space whose dimension is equal to the number of features chosen to represent the raw data. Similarly, the target space represents the ensemble of all the possible outputs, but depending on its cardinality it is possible to distinguish the two most common ML tasks: *regression* and *classification*.

Regression and classification When we talk of regression we mean that the model's output takes values in a continuous space: $\mathbf{Y} \subset \mathbb{R}^M$. Typical regression problems are the prediction of the stocks price of some company or the estimation of a house price. On the other hand, when we talk of classification, we intend that the system has to associate the input to one or more predefined *labels* or *classes*. We

notice, however, that classes cannot be a full description of the input, so they often aim to categorize it depending on the information required for a particular context. For example a computer store may be interested in knowing if an electronic device represented in a picture is a computer or not (boolean class), whereas a generic electronics store might be interested in knowing what the device actually is, e.g. a microwave oven, a fridge or a computer.

In the most general classification case, known as *multi-label classification*, each input can be matched to one or more classes in a set of C possible classes, each associated with an integer number from one to C . An example of this scenario is polyphonic sound event recognition, in which the model's objective is to detect multiple sound sources active at the same time, e.g. in a rock musical track. If, following the given example, such system detects only a bass guitar's activity it will output only its corresponding number, whereas if a bass guitar and drums are concurrently active (and detected) it will output two numbers. Due to the variable number of classes that can be detected between different inputs it is common to represent the system output as a binary one-hot array. This process is done by associating each class with the j^{th} output vector's binary entry. This means that if the j^{th} class is detected, its corresponding entry is set to one, otherwise it is set to zero. Due to this, the output vector's size is now fixed to be C , and in particular we can write: $\mathbf{y} \in \{0, 1\}^C$.

The scenario addressed by this thesis consists of a simpler classification task, which is known as *multi-class classification*. In multi-class classification the input is to be associated with only one of the C possible classes. As introduced in Chapter 1, in ASC the system is bounded to detect a single class at a time, since a single recording can only have been made in one location. Due to this, if we identify with c the class detected by the system, only the c^{th} output vector's entry will be set to one:

$$\begin{cases} y_j = 1 & \text{if } j = c, \\ y_j = 0 & \text{if } j \neq c. \end{cases} \quad (2.2)$$

Further in this chapter reasons why this output representation is particularly suitable for NNs are shown more in detail.

Minimization problem and cost function Evidently, parameters of the function introduced in Eq. (2.1) will vary depending on the function “nature”. For example, parameters describing a Gaussian mixture model (GMM) are the mean, variance and mixture weight of each Gaussian. Therefore, the aim of a SL algorithm is to find the optimal set of parameters by solving a minimization problem.

If this problem is “convex” it means that it is possible to find a closed-form unique solution through, for example, the *normal equations* method. Unfortunately

this technique is very computationally expensive since it involves the inversion of matrices that can be very large and sparse. Computational optimizations –i.e. the Cholesky decomposition— and specialized algorithms [14] have been studied, but none of them results to be applicable if the problem does not show a unique solution, i.e. if it is a “non-convex” problem.

When dealing with a very high number of parameters or with non-linearities, the problem usually does not have a unique solution and the most common approach to solve it is to gradually update an initial set of parameters $\hat{\boldsymbol{\theta}}$, usually randomly initialized, following a minimization rule. These iterative approaches rely on a comparison made between the model predictions $\hat{\mathbf{y}}^{(o)} = f(\hat{\boldsymbol{\theta}}, \mathbf{x}^{(o)})$ and the desired targets $\mathbf{y}^{(o)}$. This comparison is commonly based on the computation of a “distance” between $\hat{\mathbf{y}}^{(o)}$ and $\mathbf{y}^{(o)}$, which takes the name of *error function*, and it is here indicated with $\text{err}(\hat{\mathbf{y}}^{(o)}, \mathbf{y}^{(o)})$. Two of the most common error functions are:

- *Squared Error*:

$$\text{err}(\hat{\mathbf{y}}^{(o)}, \mathbf{y}^{(o)}) \equiv \|\hat{\mathbf{y}}^{(o)} - \mathbf{y}^{(o)}\|^2. \quad (2.3)$$

This function is typically used for real-valued and not ranged outputs.

- *Cross Entropy*:

$$\text{err}(\hat{\mathbf{y}}^{(o)}, \mathbf{y}^{(o)}) \equiv \mathbf{y}^{(o)} \cdot \ln(\hat{\mathbf{y}}^{(o)}) + (1 - \mathbf{y}^{(o)}) \cdot \ln(1 - \hat{\mathbf{y}}^{(o)}). \quad (2.4)$$

This function is used when both the output and the target take values in the range $[0, 1]$, which happens if outputs are probabilities.

Based on the error function we can define a new function $J(\hat{\boldsymbol{\theta}})$, named *cost function* or simply *cost*. This function is defined as the average of errors calculated for N training samples, that is:

$$J(\hat{\boldsymbol{\theta}}) = \frac{1}{N} \sum_{o=1}^O \text{err}(f(\hat{\boldsymbol{\theta}}, \mathbf{x}^{(o)}), \mathbf{y}^{(o)}). \quad (2.5)$$

In Eq. (2.5), $f(\hat{\boldsymbol{\theta}}, \mathbf{x}^{(o)})$ have been used instead of $\hat{\mathbf{y}}^{(o)}$ in order to highlight the dependence on $\hat{\boldsymbol{\theta}}$ of the cost. Hence, the optimization algorithm will rely on the cost to find the optimal set of parameters $\boldsymbol{\theta}$; that is:

$$\boldsymbol{\theta} = \arg \min_{\hat{\boldsymbol{\theta}}} J(\hat{\boldsymbol{\theta}}). \quad (2.6)$$

Gradient descent The most common minimization technique used to deal with Eq. (2.6) is *gradient descent* (GD). When talking of GD it is common to associate the cost to an *error surface* located in a multidimensional space, referred

as *parameter space*. This space is a dense space whose dimensionality corresponds to the number of model’s parameters and in which each point is a particular parameter configuration. Figuring this, GD defines a criteria according to which we can move, step by step, towards one of the surface minima following the steepest path. The steepest direction is found according to the derivative of the cost with respect to each of the parameters in $\hat{\theta}$, i.e. its gradient. Hence, the update of each parameter will be proportional to the value of the derivative itself. Assuming that $\hat{\theta}$ is a vector and θ_k is an element of this vector, this can be written as:

$$\Delta\theta_k \propto \frac{\partial J}{\partial \theta_k}, \quad (2.7)$$

where Δ represents the variation of the parameter θ_k from one time step to the next. This notation can be easily extended to multidimensional tensors.

As anticipated, for non-convex problems the error surface will generally show many local maxima and minima and, according to Eq. (2.6), the optimal solution of the minimization problem is represented by the global minimum. We can notice, however, that GD is by definition a criteria to move towards the closest minimum, not the lowest. According to this, we will likely reach a local minimum, not the global. It is thanks to works like [15, 16] that we can consider this not to be a problem. In these works authors explore and prove that, when dealing with a high number of parameters (as it is in NNs), there is usually no significant difference between the cost value — and so the model performance — if the algorithm reaches a local rather than the global minimum.

Underfitting, overfitting and generalization When applying a minimization algorithm there is no assurance that the cost value will continue decreasing after each iteration. Most commonly this is due to a too poor representation capacity of the chosen function, and this problem is known as *underfitting*. Since the choice of the function f is entrusted to us, the most common solution to this problem is to augment the model capacity by choosing a more complicated function f with, for example, a higher number of parameters.

On the other hand, assuming that the minimization algorithm will converge, we will obtain a function that correctly matches a given set of inputs to their expected outputs; but, since the training set is a limited representation of the entire feature space, there is no assurance that this function will correctly treat unseen inputs. This problem is widely known as *overfitting* and it can be seen as if the model has been learning all the input-output associations “by heart” rather than “understanding” them. A proper fitting of the data would consist of extracting a representation which could make the model able to act properly also on unseen samples of the

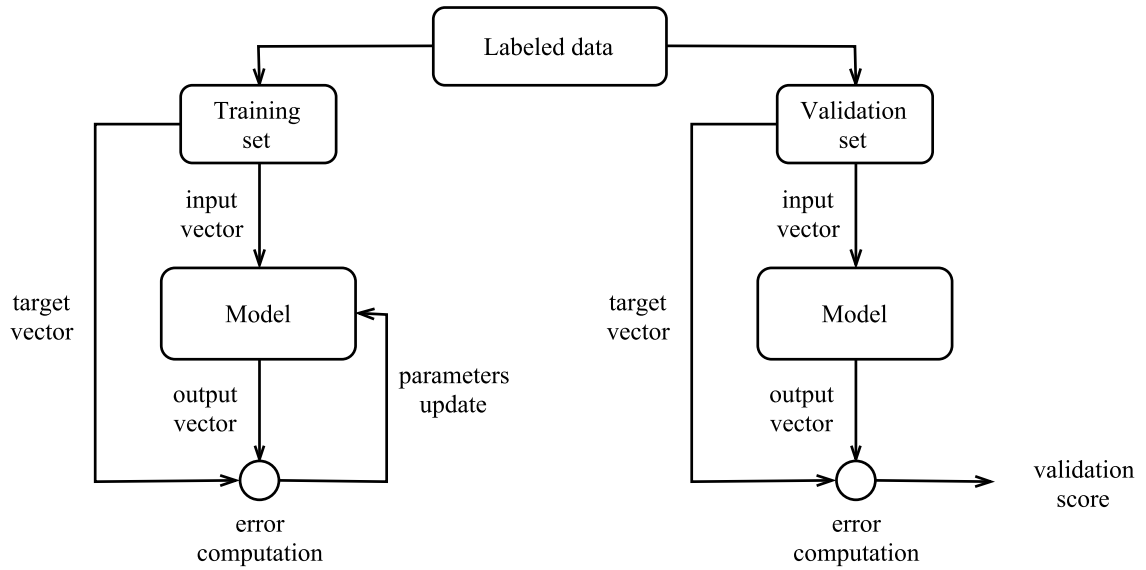


Figure 2.1: The labeled data is divided into training (left) and validation (right) data. The two sets are used respectively to update the parameters and to test the generalizing performance.

feature space. In one word it should be able to *generalize*.

The generalizing ability of the model is our very final interest and in order to understand if the model is overfitting or not, a very simple, yet effective technique may be used. This technique is known as *early stopping* and its benefits have been explored in many works, like for example in [17]. Early stopping consists of stopping the model training according to a pre-defined criteria which measures how the system is behaving in presence of unseen inputs. The most common way to perform this is by checking the model performance on a set of labeled data which are not used for training. In doing so it is usual to split all the available labeled data into two different sub-sets: a training and a validation set. Hence, the role of the validation set is to test the model generalizing performance over a set of unseen data, not influencing the parameters update. This concept is further explained in Figure 2.1.

2.2 Neural networks

In this section we introduce and describe artificial NNs retracing the most important steps that led them to be conceived and designed as nowadays. In addition, we give here a mathematical description of their parameter update algorithm and some of its most common implementations.

The objective of a NN is no different from other models', i.e. to approximate a function; but in doing so NNs have found their fundamental inspiration in the brain's structure. All NNs are built up of simple computational units which are

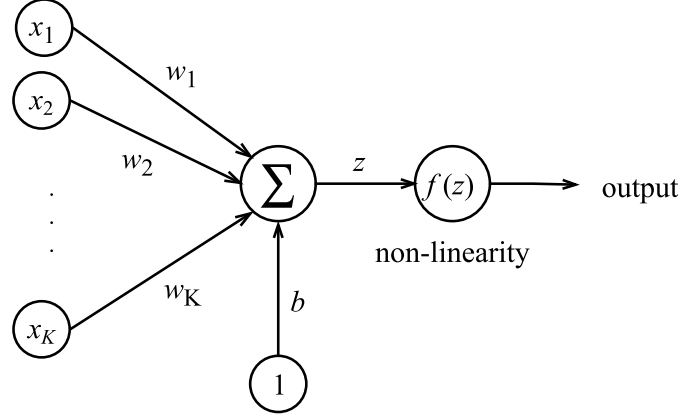


Figure 2.2: Simplified neuron scheme. All the inputs to the neuron are weighted and summed; then the output is calculated from the activation function.

densely interconnected and exchange information between them. We call these units *neurons*.

2.2.1 The neuron

Firstly introduced by Frank Rosenblatt in 1958 [18] as “perceptron”, what we nowadays call neuron is a non-linear computational unit whose schematic representation is reproduced in Figure 2.2. The figure shows that the neuron is connected to an ensemble of inputs x_1, x_2, \dots, x_K and to a bias b , from which it computes and outputs a single value. In doing so it performs a very simple two-steps computation: firstly it executes a linear combination of its inputs, weighting each of them with a different value, then it computes the output based on an *activation function*.

The first mathematical definition of perceptron is given by the following equation:

$$f(z) = \begin{cases} 1 & \text{if } z > 0, \\ 0 & \text{if } z < 0, \end{cases} \quad (2.8)$$

where

$$z = \mathbf{x}^T \cdot \mathbf{w} + b, \quad (2.9)$$

is the input to the activation function.

In Eq. (2.9) the linear combination has been vectorized, i.e. written as a dot product (\cdot) between the transposed input vector \mathbf{x}^T and the weight vector \mathbf{w} . Since they are usually written as column vectors, the transposition of \mathbf{x} is required to perform the vector multiplication. The weight vector is what characterizes the input-output function: it contains the real-valued adjustable parameters that will be modified by the training algorithm, known as *delta rule* [19]. However, Eq. (2.8) shows only one

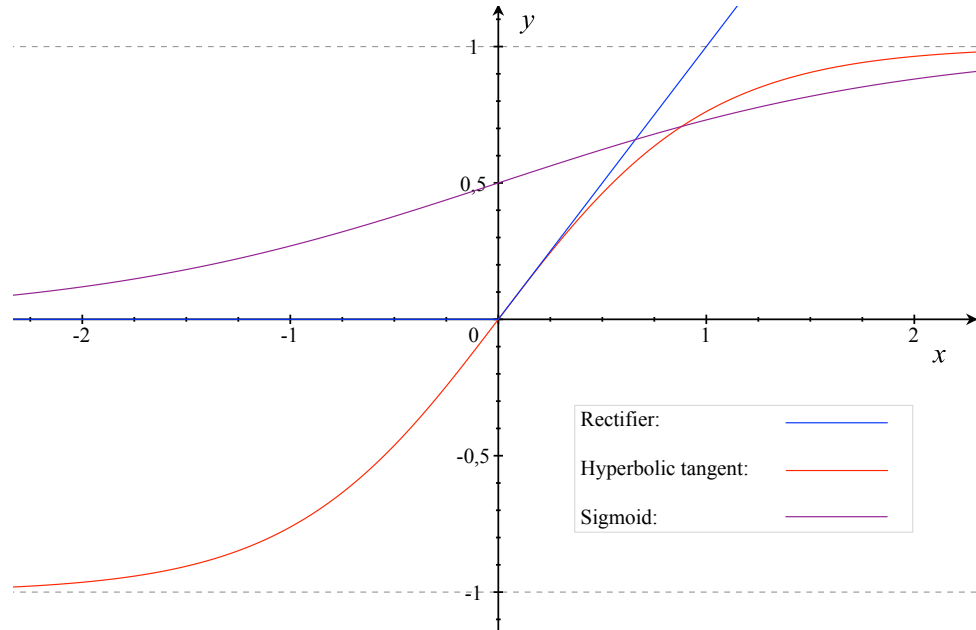


Figure 2.3: The labeled data is divided into training (left) and validation (right) data. The two sets are used respectively to update the parameters and to test the generalizing performance.

of the possible definitions of a neuron, where a step activation function is used.

In the late fifties, when these calculi were implemented by machines and the weight update was performed with electric motors, the step activation function was the most sensible function to be modelled. Nowadays, when applying a GD minimization method, properties of the activation function's derivative have to be carefully considered. The step function shows a zero-valued derivative in all its domain, except from zero, where the derivative is not defined. This property would “kill” a GD-based method. For these reasons a wide variety of activation functions has been introduced over the years, and some of them are here described.

- *Logistic function*

$$\sigma(z) = \frac{1}{1 + e^{-z}}. \quad (2.10)$$

Similarly to the step function, the logistic function, also called *sigmoid*, outputs values strictly bounded in the range $[0, 1]$. Given its smoothly growing behaviour, this function has the desirable property of being differentiable in all its domain.

- *Hyperbolic tangent*

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}. \quad (2.11)$$

The hyperbolic tangent outputs values in a limited, yet wider range than the sigmoid function: $[-1, 1]$. This function shows derivatives that can reach

higher values than the sigmoid’s derivatives. For this reason the hyperbolic tangent has proved to enhance the minimization algorithm performance during training with respect to the use of the logistic function [20].

- *Rectifier*

$$\text{Rect}(z) = \max(0, z). \quad (2.12)$$

The rectifier is an activation function that has been investigated for the first time in 2011 [21]. Neurons featuring this activation function are commonly called *rectified linear units* (ReLUs). When the input to the rectifier is lower than zero, the ReLU will output zero itself. When this happens we say that the neuron is “not active”. This property results very appealing for computational reasons but it also gives the network the ability to vary its size, depending on how many neurons are active. In addition, the rectifier does not raise any saturation problem, since it is not limited on the right side of its domain. However, this function is not differentiable when $z = 0$, so some slightly modified functions have been proposed over the years, e.g. the *SoftPlus* function [21].

2.2.2 The perceptron’s limit

Referring to Eq. (2.8), we can think of the perceptron as a classifier able to “draw” a line in the feature space \mathbf{X} , therefore subdividing it in two subspaces. The bias term is added to the linear combination so to shift the separation line from always going through the origin. Every “point” (e.g. input array) of \mathbf{X} will therefore be assigned to a zero or a one whether if it belongs in one subspace or in the other. With such function, if the weights are correctly learned, it is possible to solve some particular binary classification task, but not all of them. The issue we are going to describe is also known as the “XOR problem” and it shows the perceptron’s intrinsic weakness that opened the way to the developing of more complicated architectures, i.e. NNs.

The XOR function takes exactly two arguments, so we can imagine the feature space as a bi-dimensional plane, as shown in Figure 2.4. In this space, following the truth table of the XOR function, one of the two classes has to be assigned to the couples $(x_1 = 0, x_2 = 0)$, $(x_1 = 1, x_2 = 1)$, whereas the other class should be assigned to the couples $(x_1 = 0, x_2 = 1)$, $(x_1 = 1, x_2 = 0)$. In the figure, the two distinct classes are represented with white or black circles. One possible separation line has been drawn, but it clearly fails in separating the black from the white circles, as all possible lines would. This means that it is not possible to find a line which separates the feature space into a subspace with one output and another subspace with the other. Hence the two classes are said to be *non-linearly separable*. Because of this, for many years the perceptron has been retained a powerful but limited classifier.

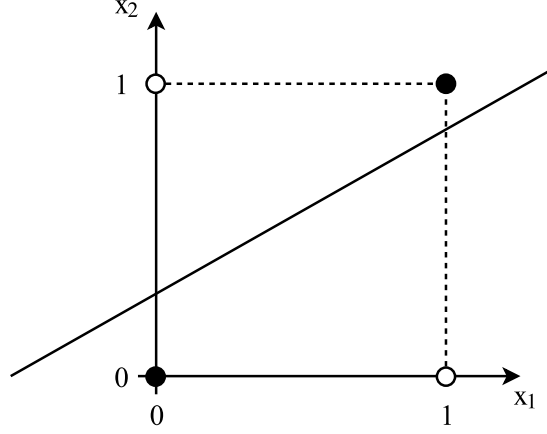


Figure 2.4: Non-linearly separable classes for the XOR problem example. The line represented fails to separate white from black circles.

The idea of more sophisticated structures — i.e. feed-forward neural networks (FNNs) — was then already known to the scientific community, but the lack of an optimal learning algorithm made it almost impossible to train such a high number of parameters. A few years later, due to the introduction of the back-propagation algorithm in 1987, the interest towards FNNs raised again.

2.2.3 Feedforward neural networks

A FNN collects a variable number of neurons arranged in a layered structure. In these networks the information flows only in one direction, from one layer to the subsequent, therefore we can represent FNNs as directed layered graphs. The feature vector commonly represents the *input layer* of these graphs, whereas the last layer is named *output layer*. All layers in between are called *hidden layers*. Each layer is composed of neurons that take inputs from the previous layer and propagate their outputs to the following. This relation can be represented as follows:

$$o_i^{(n)} = f \left(\sum_{k=1}^K w_{k,i}^{(n)} o_k^{(n-1)} + b_i^{(n)} \right). \quad (2.13)$$

Here the output $o_i^{(n)}$ of neuron i in layer n is given by its activation function (f : usually a rectifier) calculated over the weighted sum of the outputs $o_k^{(n-1)}$ of the previous layer. For the input layer, here indicated with $n = 0$, the following equation will generally hold:

$$o_k^{(0)} \equiv x_k. \quad (2.14)$$

As introduced in Eq. (2.9), we represent each weight $w_{k,i}^{(n)}$ as an entry of a weight vector $\mathbf{w}_i^{(n)}$. When dealing with FNN, it is common to group all these vectors by

stacking them as rows of a single *weight matrix* $\mathbf{W}^{(n)}$. This notation allows us to more simply refer to all parameters of layer n by straightforwardly recalling its weight matrix.

For multi-class classification problems, the output layer dimension — i.e. its number of neurons — often matches the number of possible classes C . This makes it possible to take advantage of a one-hot encoding, performed as in Eq. (2.2). By doing so we create a one-to-one correspondence between the activation of the i^{th} output neuron and the i^{th} class. In these situations it is common to use a different activation function for all neurons of the output layer, i.e. the *softmax* function:

$$\text{softmax}(z_i^{(N)}) = \frac{e^{\frac{z_i^{(N)}}{\tau}}}{\sum_{j=1}^C e^{\frac{z_j^{(N)}}{\tau}}}. \quad (2.15)$$

When using this function all neuron outputs in the last layer will exhibit the property to sum up to one, that is:

$$\sum_{i=1}^C \text{softmax}(z_i^{(N)}) = 1. \quad (2.16)$$

Therefore the softmax makes it possible for the network to output a *posterior probability distribution* among all possible classes, so that the class with the highest probability will correspond to the predicted one.

The τ parameter is known as *temperature* of the function. It is usually set to $\tau = 1$, but in doing so the function will strongly separate the highest probability from the others. If we wish to look at the network’s “certainty” it is possible to set this parameter to higher numbers, therefore reducing the gap from the highest output and the others.

Multilayer perceptrons Multilayer perceptrons (MLPs) are a particular category of FNN in which all neurons of one layer are connected to all neurons of the following. This means that, referring to Eq. (2.13), each neuron of the current layer will take as many inputs as the number of neurons in the previous layer. In the early seventies a series of studies [22] proved that MLPs could tackle the limit of the single perceptron with non-linearly separable classes. Almost twenty years later, with the publication of the *universal approximation theorem* [23], it has been proved that a MLP with one hidden layer and a proper number of neurons could represent all possible functions. However this theorem does not give any hint about the effective dimensionality of such single hidden layer, whose optimal number of neurons, for some problems, may theoretically be close to infinity. This is why it has become

common to implement architectures with multiple hidden layers, i.e. deep neural networks.

The idea behind stacking multiple hidden layers is to create a network that will learn more and more complex representations of the input features, therefore significantly improving the network representation capacity. In addition, relying on the network for learning higher level features will allow us to use low-level feature representations, therefore cutting down the need for complicated pre-processing procedures. Unfortunately, the use of a low-level feature representation has a downside. This issue, known as *curse of dimensionality* (firstly described in [24]), can be understood if we realize that low-level feature vectors have to belong to very high-dimensional feature spaces, therefore, in order to have a significant representation of such spaces, many training samples will be needed. If too few samples are provided, the network will not have a proper overview of the whole feature space, meaning that it will not be able to learn significant representations from the training data. In this case the network will likely overfit the data.

Full connectivity between neurons results in the characteristic of each of them to recognize a higher level feature when the corresponding pattern appears in the neuron’s input vector. On the one hand this means that each neuron will have its unique role, therefore contributing to enrich the network capacity, but on the other hand the recognition of a particular pattern will be correctly performed only if it appears in the same position and scale across the input vector. Especially in image recognition tasks, this will mostly represent a problem, since it is very likely, for example, that the same object will appear in different positions and sizes in two different pictures. If this object is characterized by a particular pixel pattern —e.g. it has a round shape— we would like to recognize this pattern no matter if it appears in the center or in a corner of a picture. The property of a model to correctly detect the same feature in (apparently) different inputs is called *viewpoint invariance*. Viewpoint invariance is achieved if the model is able to recognize specific patterns under many degrees of freedom, like for example shift, rotation and scaling. A degree of freedom can generally be every “transformation” that modifies the input without making it completely lose its characteristics. A possible way to achieve invariance against particular degrees of freedom is to find proper designing or normalizing strategies to apply to the input features. However, these pre-processing steps are mostly difficult to design and this is why convolutional neural networks are considered an appealing solution to this problem.

Convolutional Neural Networks CNNs find their theoretical roots in the late sixties, when studies about animals’ visual cortex [25] brought light on its structure and behaviour. These studies showed that the visual cortex is composed

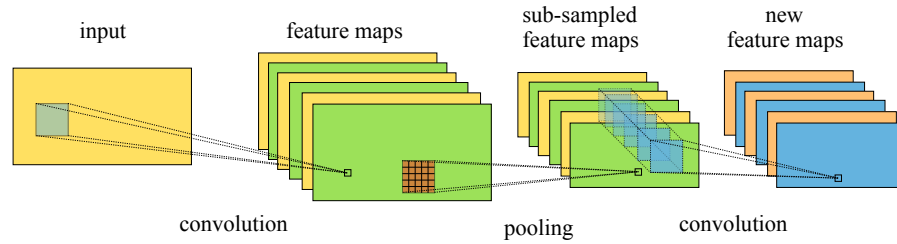


Figure 2.5: Representation of input processing in the first layers of a convolutional neural network.

of ensembles of cells in which each of them is responsible for detecting light in small areas of an image, namely the cell's *receptive field*. So it does not surprise that CNNs have been firstly applied in the field of image recognition and have become the state-of-the-art architecture when dealing with computer vision [26]. CNNs are a subcategory of FNNs and they are typically formed by the subsequent stacking of *convolutional* and *pooling* layers. Neurons in these particular layers show two important characteristics:

- *Local connectivity*: each neuron processes a *small region* of the previous layer's output, i.e. its receptive field (RF).
- *Shared weights*: in convolutional layers neurons can be collected in groups in which they all share the same weights and biases. Hence, each couple of sharing-weights neurons in the n^{th} layer will obey to the following equation:

$$\mathbf{w}_i^{(n)} = \mathbf{w}_h^{(n)}. \quad (2.17)$$

Due to the sharing-weights property it has become more common to look at each ensemble of neurons satisfying Eq. (2.17) as a single filter, called *kernel*. Therefore, each kernel slides along the input performing subsequent filtering operations; its weight matrix being the shared weight matrix itself. We define *stride* the parameter that determines how much a kernel — and so its RF — will shift from one filtering operation to the next. If, on the one hand, a small stride will allow to achieve higher invariance, on the other hand it will increase the number of filtering operations, therefore slowing the network forward pass. In Figure 2.5 a kernel's RF is represented as a light blue portion of the previous layer's output. Typically the *area* and the *stride* of a RF are free parameters that have to be designed and tested in order to find the optimal net configuration. It is common to modify Eq. (2.13) so to adapt

it to the new representation of the input:

$$o_i^{(n)} = f \left(\sum_{d=1}^D \sum_{l=1}^L \sum_{h=1}^H w_{d,l,h,i}^{(n)} o_{d,l,h}^{(n-1)} + b_i^{(n)} \right), \quad (2.18)$$

where L , H and D are the width, height and depth of the i^{th} kernel's RF. In the first convolutional layer the depth is equal to the input's, and to understand this we consider the example of a coloured picture. A coloured picture is a two-dimensional pixel matrix where each pixel is composed of three colour channels, i.e. red, green and blue. Therefore the picture has to be represented as a three-dimensional tensor of real numbers by splitting each pixel in its RGB channels, hence giving $D = 3$. For simpler problems it is possible to deal with a two-dimensional matrix as input, therefore we will have $L > 1$, $H > 1$ and $D = 1$.

Based on Eq. (2.18), outputs coming from each kernel are collected in its corresponding *feature map*. Therefore the output of the convolutional layer is a collection of feature maps whose number is equal to the number of kernels in the layer. In addition, as we move to deeper convolutional layers, the RF's third dimension D will be equal to the number of feature maps outputted from the previous layer. This is better shown in Fig. 2.5 in correspondence of the second convolution operation.

Kernels have only a small overview of the whole input, given by their RFs. The stacking of subsequent convolutional layers would make them "see" wider input's portions with a very small overview increase from layer to layer. Due to this, *pooling* layers are usually placed after each (or a few more, in very deep architectures) convolutional layer. Pooling layers operate among non-overlapping areas associating one scalar to each of them, i.e. they pool one area into one point. The most common pooling operation is *max-pooling* and it is performed by extracting the highest value from the area.

Finally, we can understand that the repetition of well-designed convolutional and pooling layers is the fundamental characteristic of CNNs. Due to these layers CNNs can achieve a complete overview of the input with good invariance to patterns shifts, hence making CNNs a powerful tool.

2.2.4 Backpropagation algorithm

Basics of the backpropagation (BP) algorithm have been known since the early sixties [27]. Despite this fact, it was only in the late eighties that it became the standard learning algorithm for NNs due to some experiments conducted by Rumelhart et al. [28]. In this work they investigate and empirically demonstrate the emergence of useful internal representations in the network's hidden layers after it was trained with the BP algorithm.

BP is a method for computing all gradients that will be used for the update of each neuron’s weights. This calculation is based on the minimization of a cost $J(\hat{\boldsymbol{\theta}})$ performed with a GD algorithm (see Section 2.1) coupled with the *derivative chain rule* which is used to “propagate” the cost derivatives within the network’s hidden layers. If we want to summarize the BP algorithm it is possible to split it into two fundamental steps:

- *Forward pass*: an input is applied to the first layer and propagated through the network, so that all activations for all neurons are computed.
- *Backward pass*: based on the desired target output, derivatives of the cost function with respect to the current output are back-propagated from the last layer to the first.

In order to give a mathematical description of BP we start by defining the basic gradient-based update rule for each of the NN’s parameters:

$$w_{k,i}^{(n)}(t+1) = w_{k,i}^{(n)}(t) - \frac{\eta}{B} \cdot \sum_{o=1}^B \frac{\partial J_o(\mathbf{W})}{\partial w_{k,i}^{(n)}(t)}, \quad (2.19)$$

where t indicates the current time step and η is usually called *learning rate*. Its typical value is $\eta \approx 10^{-3}$, since it determines the magnitude of the weight updates at each step. Furthermore \mathbf{W} , used here in place of $\hat{\boldsymbol{\theta}}$, is a tensor that collects all network parameters $w_{k,i}^{(n)}$. It is straightforward to deduce that a “slice” of this tensor is the weight matrix $\mathbf{W}^{(n)}$ of layer n , as it was introduced after Eq. (2.13). The bias terms $b_i^{(n)}$ are here omitted since they can be seen as weights acting on inputs fixed to one. According to this, it is possible to insert each bias in its corresponding weight vector $\mathbf{w}_i^{(n)}$ as its 0th entry. This operation can be written as:

$$\begin{cases} w_{0,i}^{(n)} = b_i^{(n)}, \\ o_0^{(n-1)} = 1. \end{cases} \quad (2.20)$$

$$(2.21)$$

Moreover, in Eq. (2.19) we use $J_o(\mathbf{W})$ to indicate the value of the cost function calculated for one training example o out of a batch of B samples ($B \leq O$). Since \mathbf{W} is the cost’s only dependency, it is possible to simplify the notation by writing only J instead. In addition, we will always refer to a single training sample hereafter, therefore omitting also the sample index o . Finally, since all the following considerations will be conducted for a fixed time step t , we will omit all time dependencies as well.

To evaluate the partial derivative in Eq. (2.19) it is possible to apply the derivative

chain rule as follows:

$$\frac{\partial J}{\partial w_{k,i}^{(n)}} = \frac{\partial J}{\partial z_i^{(n)}} \frac{\partial z_i^{(n)}}{\partial w_{k,i}^{(n)}}, \quad (2.22)$$

where, as introduced in Eq. (2.9), $z_i^{(n)}$ is the input to the i^{th} neuron's activation function in layer n . Thanks to Eq. (2.20) we can rewrite Eq. (2.9) as:

$$z_i^{(n)} = \sum_{k=0}^K w_{k,i}^{(n)} o_k^{(n-1)}, \quad (2.23)$$

where, in addition to the bias omission, the vector product is now written in its explicit form.

Given the basic derivative properties, we can easily differentiate the quantity expressed in Eq. (2.23) and obtain:

$$\frac{\partial z_i^{(n)}}{\partial w_{k,i}^{(n)}} = o_k^{(n-1)}. \quad (2.24)$$

Therefore, Eq. (2.22) will reduce to:

$$\frac{\partial J}{\partial w_{k,i}^{(n)}} = o_k^{(n-1)} \frac{\partial J}{\partial z_i^{(n)}}, \quad (2.25)$$

where the partial derivative of the cost with respect to $z_i^{(n)}$ is usually called *error* of the i^{th} neuron in layer n . Applying the chain rule once more and recalling Eq. (2.13) we obtain:

$$\frac{\partial J}{\partial z_i^{(n)}} = \frac{\partial J}{\partial o_i^{(n)}} \frac{\partial o_i^{(n)}}{\partial z_i^{(n)}} = \frac{\partial J}{\partial o_i^{(n)}} f'(z_i^{(n)}), \quad (2.26)$$

where f' is the first derivative of the activation function. Given this last identity we can finally write:

$$\frac{\partial J}{\partial w_{k,i}^{(n)}} = o_k^{(n-1)} f'(z_i^{(n)}) \frac{\partial J}{\partial o_i^{(n)}}. \quad (2.27)$$

Eq. (2.27) gives a useful expression for the cost derivative with respect to each weight, which is the quantity needed in Eq. (2.19) for the weight update. So, in order to compute this quantity for all weights of one neuron, we must be able to calculate the cost derivative with respect to that neuron's output. This derivative is easy to calculate if the neuron belongs to the last layer of the network, since, as shown in Eq. (2.6), the cost directly depends on all the outputs of the last layer. This means that Eq. (2.27) is easily computable if $n = N$. In order to compute the cost derivative with respect to a generic hidden neuron's output — e.g. $o_k^{(n-1)}$ — we must consider that that output will influence all neurons inputs $z_i^{(n)}$ in the next

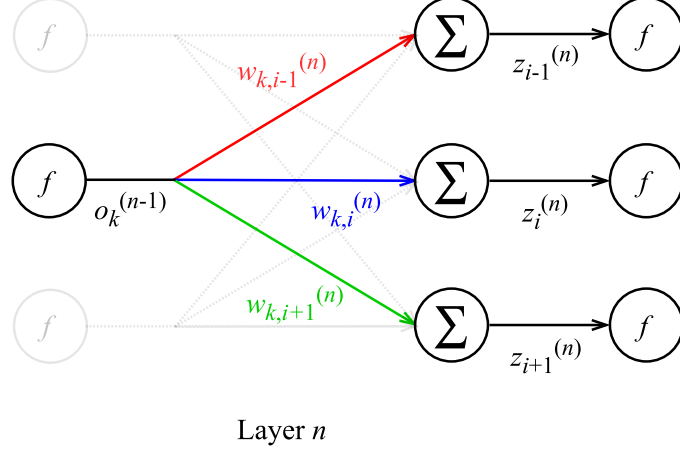


Figure 2.6: Propagation of a generic neuron output. Here we show how the output $o_k^{(n-1)}$ influences all neurons in the following layer.

layer, proportionally to each weight connecting them. This is better shown Fig. 2.6. As a result of this consideration we can look at the desired derivative as a sum of I terms, where I is the number of neurons influenced by that specific output:

$$\frac{\partial J}{\partial o_k^{(n-1)}} = \sum_{i=0}^I \frac{\partial z_i^{(n)}}{\partial o_k^{(n-1)}} \frac{\partial J}{\partial z_i^{(n)}} = \sum_{i=0}^I w_{k,i}^{(n)} \frac{\partial J}{\partial z_i^{(n)}}. \quad (2.28)$$

In Eq. (2.28), the latter term is obtained considering the expression for $z_i^{(n)}$ given in Eq. (2.23). Hence, Eq. (2.28) shows that errors computed for each neuron in layer n can be linearly combined to give the derivative of the cost with respect to each output of the previous layer. Considering Eq. (2.26) and Eq. (2.28) we can finally rewrite the latter as:

$$\frac{\partial J}{\partial o_k^{(n-1)}} = \sum_{i=0}^I w_{k,i}^{(n)} f'(z_i^{(n)}) \frac{\partial J}{\partial o_i^{(n)}}. \quad (2.29)$$

This relation is the key of the BP algorithm since it binds the cost derivatives with respect to the outputs of two adjacent layers. So, by coupling Eq. (2.27) and Eq. (2.29) it will be possible to compute the weight updates not only for neurons in the last layer, but for the whole the network. This will be possible by simply backpropagating the cost derivatives with respect to the outputs from one layer to the other, starting from the last.

Stochastic gradient descent and momentum When training a NN it is commonly preferable to have big amounts of training data. If the amount of data is big enough to cover the most of the feature space, the network will more likely be able to learn meaningful higher level features. A naive GD-based update requires

that, for a correct weight update, the update itself is performed only when all the training data has been “seen” by the network. This is called *full-batch learning* and it corresponds to having $B = O$ in Eq. (2.19) (we remind that O has been introduced as the total number of training samples). However, when O is a large number, a full-batch approach may represent a problem due to computational and memory issues. Because of this it can be preferable to update the parameters before all the training data has been used. This method, known as stochastic gradient descent (SGD), consists in accepting to calculate the cost (and its derivatives) on a smaller batch of training data, therefore giving us stochastic approximations of these quantities.

The most extreme application of SGD, known as *on-line learning*, consists in consequentially updating the net parameters at each sample of the training data, thus having $B = 1$. This technique leads to very bad approximations of the cost gradient, so, if possible, it is often avoided. A more common and soft approach is to update parameters after a more consistent amount of training cases, and this is called *mini-batch learning*. This approach has proven to be the most sensible compromise in order to have frequent parameters updates and good approximations of the cost gradient.

The *momentum method* is a more refined technique used to calculate the weight updates, and it can be applied to enhance the SGD optimization. This technique relies on the definition of a new quantity v , named *velocity*, which is used to keep track of all gradients previously obtained at each update step. The weight update equation is therefore re-adapted as shown in the following two equations:

$$\begin{cases} v(t) = \alpha \cdot v(t-1) - \frac{\eta}{B} \cdot \sum_{o=1}^B \frac{\partial J_o}{\partial w(t)}, \\ w(t+1) = w(t) + v(t), \end{cases} \quad (2.30)$$

$$(2.31)$$

where α is a parameter usually in the range $[0.5, 1)$ and determines how slowly or quickly the previous momentums will decay. The notation has been here simplified with respect to Eq. (2.19) by taking for granted that this new update rule is applied to all the weights $w_{k,i}^{(n)}$ of the network.

Another possible variant of the momentum method is called *Nesterov momentum* and it has proven to give better results than the naive momentum [29]. This technique relies again on a velocity-based weight update, but the new velocity is calculated after the update has been performed based only on the previous velocity. The update rule therefore becomes:

$$\begin{cases} w(t+1) = w(t) + v(t), \\ v(t+1) = \alpha \cdot v(t) - \frac{\eta}{B} \cdot \sum_{o=1}^B \frac{\partial J_o}{\partial w(t+1)}. \end{cases} \quad (2.32)$$

$$(2.33)$$

By doing so it is possible to improve the velocity stability and also to obtain faster learning convergence.

2.2.5 Regularization

In the end of Section 2.1 early stopping was introduced. This, like many other techniques aiming to reduce overfitting, falls under the name of *regularization* techniques. This family of techniques addresses issues typically encountered in ML, therefore we can find that some of them have only been inherited by the field of NNs — e.g. L_1 and L_2 regularization [30]. In addition to these, also new techniques specifically designed for NNs have been introduced over the years, the most common among them being *dropout*.

Dropout Many studies — e.g. [31, 32] — have proved that combining different classifiers can result in a new classifier that outperforms the best single classifier used in the combination. This concept becomes intuitive if we think that each classifier is likely specialized in identifying particular features, meaning that a combination of the whole ensemble should show all characteristics of the different components. By different we mean either that models are trained on different datasets or that they show different architectures or parameters configurations. In both these scenarios, the difficulty of gathering different datasets or to design different optimal architectures is not to be underestimated.

Firstly introduced by Hinton in his video-course lesson — a more detailed description can be found in [33] — dropout is a technique that brilliantly manages to tackle the difficulty of training different models separately. The purpose of dropout is to train many different sub-models with lower capacities — by sub-sampling the original one — and then combine them together once the training is over. It operates by setting a probability $p^{(n)}$ for each neuron in layer n to be dropped (i.e. removed) at training time. This means that, in some epochs, some neurons do not participate to the forward pass and therefore they will not have their parameters updated in the backward pass. The amount of dropped neurons is depending on the chosen probability $p^{(n)}$, but it is still stochastic, therefore there is no way of predicting which neurons are used in one epoch and which are not.

At test time we want to use the combination of the different sub-models, therefore dropout has to be “switched-off”, letting all neurons to be active at the same time. Though, this simple switch-off is not sufficient. Let us consider a single neuron: if all neurons of the previous layer become active at the same time, we would have that its expected average input drastically changes. This is due to many more inputs active at the same time with respect to the training phase. This issue can be tackled by

scaling the weight matrices of ex-dropped layers, and it is done as follows:

$$\mathbf{W}_{\text{test}}^{(n)} = (1 - p^{(n)}) \cdot \mathbf{W}_{\text{train}}^{(n)}. \quad (2.34)$$

In Eq. (2.34) the weights $\mathbf{W}_{\text{train}}^{(n)}$ calculated during the training phase are multiplied by the retention probability $(1 - p^{(n)})$. In doing so we obtain a classifier that is a proper average of all the sub-models trained when dropout was active.

In our model dropout is used in the input connections and after each convolutional layer, with different dropping probabilities being tested. Some evaluations on the effectiveness of dropout are reported in Chapter 4.

2.3 Audio features

A raw audio digital signal is always represented in the temporal domain, where it corresponds to a discrete series of real-valued samples that form the waveform of the signal. However, even if temporal representations have recently proven [34] to give good results when using CNN in audio event recognition, they still result in a worse performance with respect to higher level representations. Therefore, in this section we will describe the steps (summarized in Fig. 2.7) that lead to the representation chosen for our work: the log-mel spectrogram.

Frequency and time-frequency representations When aiming to operate on a spectral representation of an audio signal, a transformation of the signal itself is firstly necessary, i.e. the discrete Fourier transform (DFT). The DFT allows to map any discrete signal, given its temporal evolution, into a list of complex values representing the coefficients of a linear combination of complex sinusoids. For this reason it is said that the transformed signal resides in the frequency domain and its representation takes the name of *spectrum* of the signal. If the DFT is operated over the entire signal, its spectrum will result to have a very high frequency resolution, meaning that information about very close frequencies can be represented. On the other hand, by transforming the signal in its entirety, all the temporal information about the sequentiality of the events will be lost in favour of this frequency resolution.

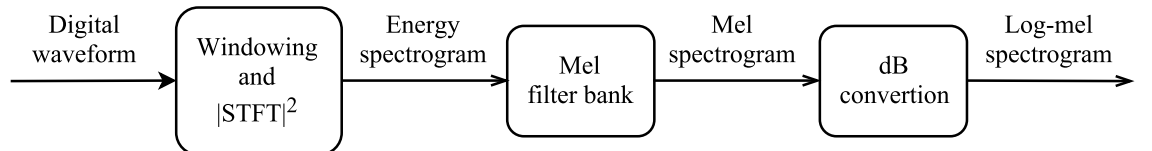


Figure 2.7: Feature extraction block diagram from digital raw data to the log-mel spectrogram. All steps are described in Section 2.3.

This is the reason why a pure frequency representation is usually avoided in favour of a hybrid time-frequency representation, i.e. the *spectrogram*. The spectrogram is a collection of subsequent segments of the raw signal, each of them transformed with a DFT. These segments are usually named *frames*. The transformation of smaller portions of the signal is made under the assumption that an audio signal is stationary — i.e. its statistical properties do not change — over temporal windows of 20-40 ms. Therefore this particular application of the DFT takes the name of short-time Fourier transform (STFT).

When operating the STFT of an audio frame it is common to multiply the frame by a window function, like for example the Hamming function. This is required since a frame will likely show discontinuities at its edges that, if not smoothed, will produce a broadband noise — i.e. additive power contributes over all frequencies — in the spectrum. A window function will therefore attenuate the signal near the edges and emphasize the central portion. Because of the windowing, we will need to take overlapping frames in order to make up for the attenuated parts of the signal. Typically chosen overlaps are 50/75%, which have been proved [35] to include up to 90% of the original data information in the calculated coefficients. After the STFT coefficients are calculated, we take their squared magnitudes in order to obtain the spectral magnitude representation of the frame.

The log-mel scale The frequency scale obtained so far is linear, meaning that all adjacent frequencies are equally distant from each other. However, psychoacoustic studies [36] proved that the human perception of sound pitches follows a logarithmic rule. Due to this, after a frequency around 500 Hz, we perceive increasingly large frequency intervals to determine equal pitch increments. Because of this, the frequency scale is usually converted into another non-linear scale, the *mel scale*. The conversion formula is the following:

$$m = 1125 \cdot \ln \left(1 + \frac{f}{700} \right), \quad (2.35)$$

where f and m respectively indicate the frequency in the linear (Hertz) and in the mel scales. The practical way to operate this conversion is by applying a mel filter bank, that is we multiply each frame's spectrum by a sequence of triangular filters, each of these filters corresponding to a mel band. These filters cover increasingly wider frequency ranges on a linear scale, whereas they will be equally spaced on the mel (logarithmic) scale. By applying this multiplication we will obtain an array of real-valued numbers for each frame, where the length of the array corresponds to the number of filters used in the mel filter bank. Typical values are 20, 40 or 60 mel bands.

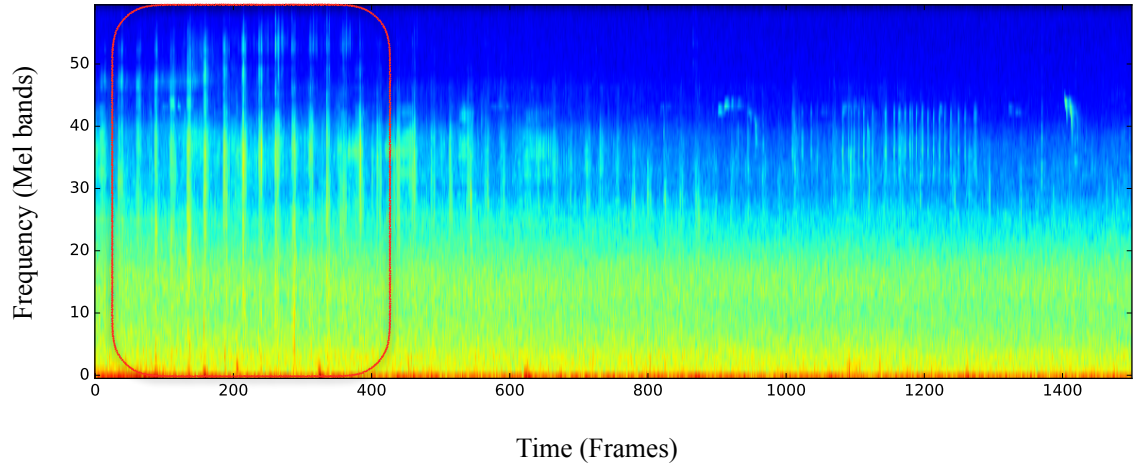


Figure 2.8: Log-mel spectrogram of a 30-second audio segment. In the beginning (circled in red) it is possible to distinguish the pattern of lines regularly separated in time and occupying almost all log-mel bins (from 0 to 22050 Hz). Those lines represent footsteps in a forest path.

In order to achieve an optimal representation a final step is still required. According to the Weber-Fechner law, humans tend to perceive external stimuli (such as sound loudness) with a logarithmic law. In other words, if we increment the energy of a sound, a human will perceive a different loudness increment depending on the initial energy value. The higher is the initial value, the lower is the perceived loudness increment. Due to this, a final processing step requires to introduce a non-linearity also among each mel band’s energy value. This is typically done by converting each energy to the logarithmic scale, therefore obtaining the *log-mel spectrum* of the frame. If we apply this procedure for each frame and then concatenate the results, we will obtain a representation similar to the one reported in Fig. 2.8.

2.4 Previous work

Firstly defined in Bregman’s book [37] in 1994, the field of “auditory scene analysis” was born to study how to model the auditory perception of both humans and artificial systems. Concerning artificial analysis, we can find the very first important contribution to the field of CASA in Wang and Brown’s [38]. The main focus of these two theoretical works is to give a proper background for future research by addressing issues at the basis of auditory scene modelling. One of the most relevant audio analysis issues, known as the “mixture” or “cocktail party” problem, concerns the recognition of one person’s speech in a mixture of different overlapping speeches. This problem has gained emphasis throughout the years because its concept can be easily extended to every mixture of sounds, going from musical notes or instruments to environmental noises.

Ten years after Bregman’s work, Divenyi’s “Speech Separation by Humans and Machines” [39] will be published as a collection of many theoretical papers concerning the future of CASA. In particular, Divenyi’s works contains Slaney’s contribution [40] which aims to give a redefinition of the field by asserting that sound separation is not the most proper way to deal with mixtures of sounds and their comprehension. In his evaluations he examines how low-level representations — like correlograms and cochleagrams — can solve the problem of separating sources, but they fail to achieve a proper high-level modelling for human perception. Years later, Peltonen’s empirical work [41] will highlight that human recognition is mostly based on the identification of prominent sound events.

Alongside this theoretical framework, also practical contributes to ASC began to appear in the same period and new contributions continued to accumulate. In Table 2.1 we report some of the most important works by showing the respective feature representations and classification systems.

The first example goes back to 1997 and it is represented by Sawhney and Maas’ contribution [42]. Here the authors’ goal is to discriminate five environmental sounds (“people”, “subway”, “traffic”, “voice”, and “other”) over a three-hours dataset. In this work, recurrent neural networks (RNNs) and nearest neighbour classifiers are used with *relative spectral*, *power spectral density*, and *frequency bands* features.

One year later, a hidden Markov model (HMM) approach has been proposed by Clarkson et al. in two different works [43, 44]. In [43] authors address the issue of recognizing different sound objects — e.g. different speakers in a multiple-speaker environment — and the detection of scene change. On the contrary, in [44] they focus on the deduction of environmental context through audio classification. Based on this latter work, Sawhney et al. [49] managed to conduct experiments with a wearable system capable to determine if the wearer was involved in a conversation or not.

In [45] authors present a system able to recognize five different types of TV

Table 2.1: Main ASC previous works.

<i>features</i>	<i>system</i>	<i>reference</i>
various features	RNN + nearest neighbour	[42]
spectral features	HMM	[43, 44]
various features	NN	[45]
various features	GMM + nearest neighbour	[46]
line spectral frequency	various classifiers	[47]
MFCC	GMM	[46]
various features	GMM + nearest neighbour	[46]
various features	HMM-GMM + nearest neighbour	[48]

programs (“commercials”, “basketball games”, “football games”, “news reports”, and “weather forecasts”) based on acoustic information. In this work a set of low-level features is used — e.g. *spectral* and *volume distribution* features — to feed a NN-based classifier.

One year later, El-Maleh et al. [47] investigated the recognition of five common mobile environments (“car”, “street”, “babble”, “bus”, and “factory”) with the use of *line spectral frequency* features. These features are tested with four different classifiers, reaching a performance peak when a quadratic Gaussian model is used for classification.

In 2002 a framework [46] comprehending GMM and nearest neighbour classifiers was implemented to recognize 26 different acoustic environments. In doing so, authors made use of many feature representations, going from mel-frequency cepstral coefficients (MFCCs) (used only for the GMM classifier) to time and frequency features.

Eronen et al. [48] approached ASC performing a comparison between GMM-HMM and nearest neighbour classifiers. The dataset contains audio files subdivided into 27 different contexts, each of which associated with six high-level categories: “outdoor”, “vehicles”, “public places”, “quiet places”, “home”, and “reverberant places”. In their work authors tested a wide variety of time, spectral, and cepstral features, with a particular focus on MFCCs and their deltas. In addition, tests with three different linear feature transformations were performed on cepstral features, proving to give a slight improvement of the recognition accuracy.

Nowadays, research in ASC is mostly pushed by the IEEE AASP TC thanks to the DCASE challenges. The first challenge took place in 2013 [50] introducing a development and an evaluation dataset of 100 30-second segments equally divided among ten different classes. Both datasets — the 2016’s will be described in Chapter 4 — are now publicly available, thus making it possible to compare new systems with those that have been proposed during challenges. Some of the best performing systems proposed for the DCASE 2013 ASC task are HMM-GMM [51], and support vector machines (SVM) [52] which all showed very good accuracy scores on the evaluation dataset. However, the highest score was reached by a SVM model trained on *recurrent quantification analysis* features extracted from MFCCs [53]. Accuracy scores reached by some of these models on the evaluation dataset are reported in Chapter 4 and compared to the system proposed in this thesis.

3. METHODOLOGY

In this chapter we describe the system we propose for the solution of the ASC task. In Fig. 3.1 the system is summarized as a chain of processing steps. After extracting audio features from the raw segment, we normalize and split them into chunks. These chunks are then fed to a CNN which outputs one prediction vector for each of them. Finally, all prediction scores are combined and the audio file's class is obtained.

3.1 Feature extraction and pre-processing

Our system is built to operate with audio segments characterized by a sampling frequency of 44.1 KHz and 24-bit resolution. In addition, mono, stereo and binaural audio channels are supported.

As first step we check if the audio file is composed of two channels; if so, we average them together into a single audio channel. After this, the audio segment is split into frames of 40 ms with 50% overlap. Each frame is then multiplied by a Hamming window and transformed with a 2048-points STFT. Then, the square of the absolute value of each coefficient is computed. After this, energies E_b within 60 different mel-bands are calculated with a mel filter bank; filters are applied with a frequency range going from zero to 22.05 KHz. Finally, as introduced in Chapter 2, we apply a dB conversion. If we consider a single frame we can define $E_{b, \text{linear}}$ as the energy in the b^{th} mel band in the linear scale, also known as *bin*. Thus, we can obtain the bin value in the dB scale $E_{b, \text{dB}}$ as:

$$E_{b, \text{dB}} = 10 \cdot \log_{10}(E_{b, \text{linear}}). \quad (3.1)$$

The whole feature extraction procedure is implemented in Python with the *li-*

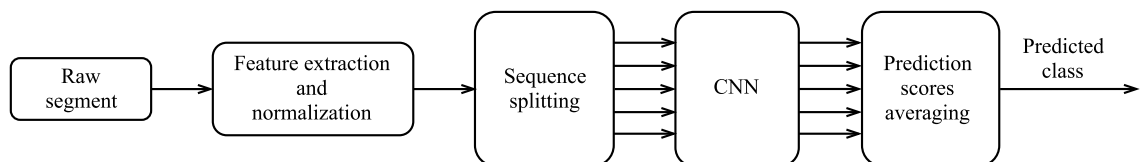


Figure 3.1: Block diagram of the proposed model: from raw data to the classification of the audio scene.

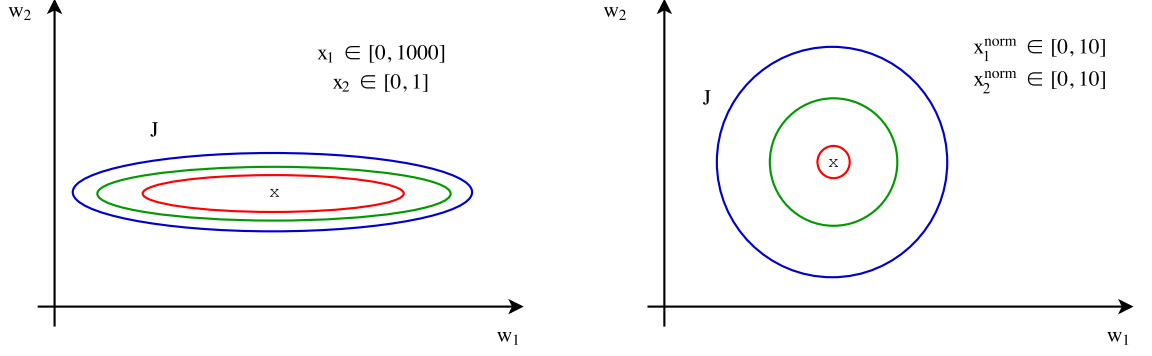


Figure 3.2: Feature normalization effect on the shape of the cost function. A simple case of a two-dimensional feature space is here considered. After the normalization the cost partial derivatives will not be influenced by the stretched shape and will point more directly to the minima.

brosa [54] audio library.

Normalization In order to speed up the network optimization algorithm it is common to normalize features so that they will show zero means and unit variances. Referring to the notation introduced in Eq. (3.1), we define the normalization formula for each bin as:

$$E_{b, \text{dB}}^{\text{norm.}} = \frac{E_{b, \text{dB}} - \mu_b}{\sigma_b}, \quad (3.2)$$

where μ_b and σ_b are the mean and the standard deviation of the b^{th} bin calculated over all training frames.

Reasons behind feature normalization are essentially two. The first reason is that too high values in the feature vector may lead some neurons — e.g. logistic, hyperbolic tangent — to work in their saturation region, hence giving birth to approximately null gradients. In addition to this, constraining different features to take values in the same range will lead to an error surface which will not be “overelongated” on some directions with respect to others. Due to this, the parameter update will move more smoothly along the surface toward the desired minimum. In Fig. 3.2 we represent a simple example of how the normalization affects the shape of the cost function in a two-dimensional feature space. Without feature normalization the GD “walk” towards the minima — i.e. the best parameter configuration — will likely be a zigzag line influenced by the different amplitudes of the two cost derivatives with respect to each of the parameters. Because of this, very slow convergence times are likely to be observed. Moreover, the number of updating steps required may be very different depending on the choice of the initial set of parameters. This fact is not desirable, especially when random weights initialization is performed.

Sequence splitting As final step of the feature processing procedure we subdivide each audio segment into non-overlapping sequences. Different performances for different sequence lengths are compared in Chapter 4. Since each segment is associated with a target vector \mathbf{y} of 15 one-hot encoded bits, we have to replicate each target vector for each sequence in which the segment is split.

3.2 Proposed neural network

The proposed CNN is designed to work on “single-channel images”, namely the normalized log-mel energy spectrogram of a sequence. The spectrogram is processed by a convolutional layer in which each RF’s stride is kept unitary in both directions. Typically used RF areas are 3×3 or 5×5 : we choose ours to be the latter since it best fits the input dimension for many sequence lengths. Then, convolutional layer’s outputs are grouped in feature maps whose dimension is the same as the input’s: this fact is due to the unitary stride and to a small amount of zero-padding that is done on the input borders.

After the convolution, a max-pooling layer is used to sub-sample each feature map. The sub-sampling is performed by using 5×5 non-overlapping max-pooling windows, therefore obtaining a reduction of both the feature maps’ height and width. The reduction is proportional to the pooling window’s area, therefore feature maps result in being 25 times smaller after being sub-sampled.

The second convolutional layer is characterized by kernels with the same RF and stride as in the first layer, but their number is higher. This is typically done in order to allow the network to increase its representation capacity as its architecture grows deeper. In both convolutional layers the rectifier function, described in Chapter 2, has been used as activation function.

After the second convolutional layer a set of new feature maps is obtained and a last max-pooling sub-sample is performed on them. This final sub-sampling is done by aiming at a complete shrinking of the temporal dimension: in Fig. 3.3 we show how the second (red) pooling window is designed to cover the whole horizontal dimension.

Finally, the last layer is composed of a set of 15 (one per class) fully connected neurons. By fully connected we mean that each neuron has its own weight vector and is connected to all inputs coming from the previous layer. The activation function used for this layer is the softmax function, therefore the effective output of the network is an array $\hat{\mathbf{y}}$ of 15 real values in the range $[0, 1]$ where the position of the highest value will indicate the class associated with the input.

Since the network is operating on a sequence-wise classification level, we had the necessity to implement a “voting” strategy so that the whole segment classification could be performed. If we let S to be the number of sequences in which the segment

is split we can define $\hat{\mathbf{y}}^{(s)}$ as the prediction vector for the s^{th} sequence. Therefore, its j^{th} entry $\hat{y}_j^{(s)}$ represents the prediction score for class j to be associated with the s^{th} sequence. The segment-wise classification is then performed by calculating:

$$j^* = \arg \max_j \left[\frac{1}{N} \sum_{s=1}^S \hat{y}_j^{(s)} \right]. \quad (3.3)$$

Therefore, the class associated with the whole segment is j^* , given that the \hat{y}_{j^*} is the maximum entry in the vector containing the average of all sequence-wise prediction scores.

3.3 Training and regularization

The loss function we use to calculate the cost and its gradients is the categorical cross-entropy, already described in Chapter 2. Cost derivatives with respect to all parameters, needed for their update, are then calculated through BP. Finally, each parameter is updated according to a recently introduced optimization algorithm, that is the *Adam* optimizer.

Adam This algorithm, introduced for the first time in 2014 by Kingma et al. [55], is a stochastic optimizer which relies on the computation of first and second order momenta of the estimated gradient, here defined as m_t and v_t , with t being the current time step. Recalling the notation introduced in Chapter 2 (Eq. (2.30)), momenta are now calculated as:

$$\begin{cases} m(t) = \beta_1 \cdot m(t-1) + (1 - \beta_1) \cdot \frac{1}{B} \sum_{o=1}^B \frac{\partial J_o}{\partial \theta(t)}, \\ v(t) = \beta_2 \cdot v(t-1) + (1 - \beta_2) \cdot \left(\frac{1}{B} \sum_{o=1}^B \frac{\partial J_o}{\partial \theta(t)} \right)^2, \end{cases} \quad (3.4)$$

$$(3.5)$$

where β_1 and β_2 are the momenta forgetting factors — usually in the range $[0.9, 1]$ — and θ is used to represent a generic network parameter. In order for momenta to have the same expectations as the gradient and its square (see [55] for more details), it is necessary to apply a bias correction to both momenta. This correction is done as:

$$\hat{m}(t) = \frac{m_t}{(1 - \beta_1^t)}, \quad (3.6)$$

$$\hat{v}(t) = \frac{v_t}{(1 - \beta_2^t)}. \quad (3.7)$$

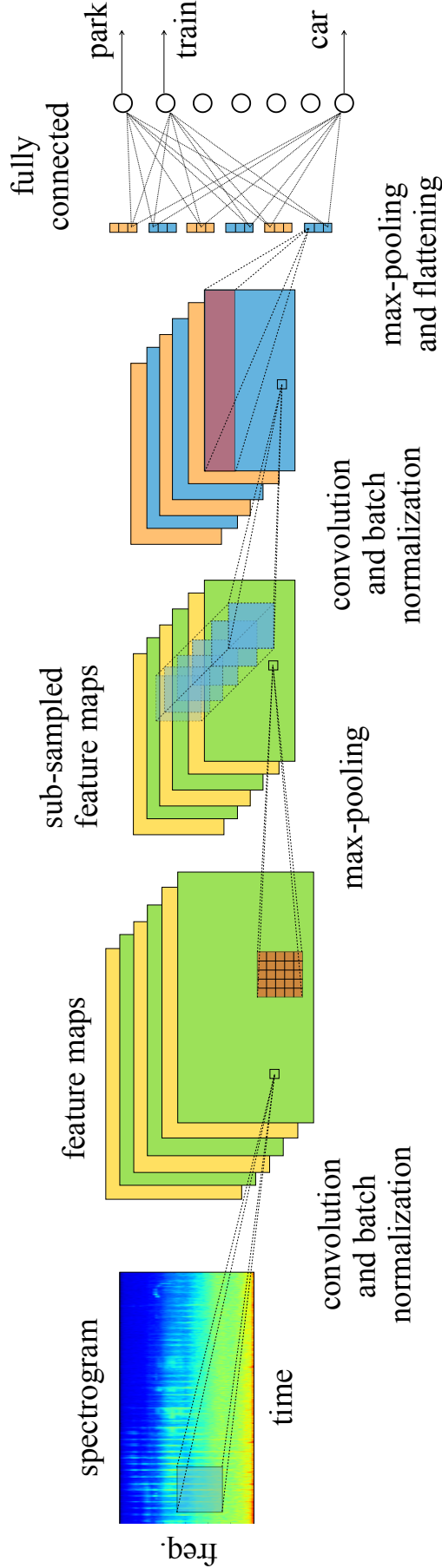


Figure 3.3: Proposed network’s architecture. Receptive fields are represented in light blue, whereas the max-pooling windows are in red.

Finally, the update rule for each network parameter is given by the following:

$$\theta(t) = \theta(t-1) - \eta \frac{\hat{m}(t)}{\sqrt{\hat{v}(t) + \epsilon}}, \quad (3.8)$$

where η is the learning rate and ϵ is a small value used in order to avoid the division by zero. The final parameter configuration used for our model is Keras' suggested parameter set: $\eta = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.99$ and $\epsilon = 10^{-8}$.

Non-full and full training The proposed training method consists of two phases: *non-full* and *full training* (Fig. 3.4). These phases are named after our choice to use or not a validation dataset for monitoring the network performance during the training phase. Following this, in the first phase, a validation dataset must be retained, therefore we choose to keep 20% of the labelled data and not to use it for training. Once segments belonging to the training and validation sets are chosen, the training can start: at each epoch we concatenate training spectrograms into class-wise feature lists so to randomly shuffle and time-shift all segment spectrograms before they are split into sequences. This is done in order to show the network always slightly different sequence spectrograms, hence increasing the input variability. Then, every five epochs we check the segment-wise performance on both training and validation sets. We save the network parameters only if the validation score has improved. If no improvements are encountered after 100 epochs, we stop the training.

By running several experiments we noticed a saturating behaviour of the segment-wise validation accuracy curves, as we show in Fig. 3.5. Due to this peculiar behaviour, we can estimate an average number of epochs needed for the convergence of the validation score on the four folds. Hence, based on this average, we decide to train the network as done in the non-full training phase, but this time on the whole

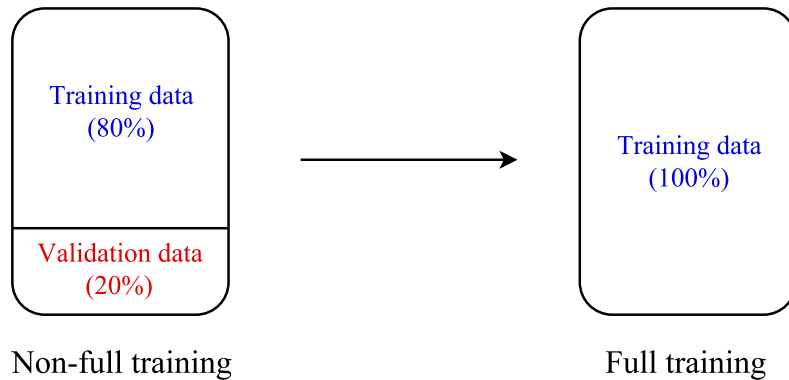


Figure 3.4: Splitting of the labelled data for the non-full and the full training phases.

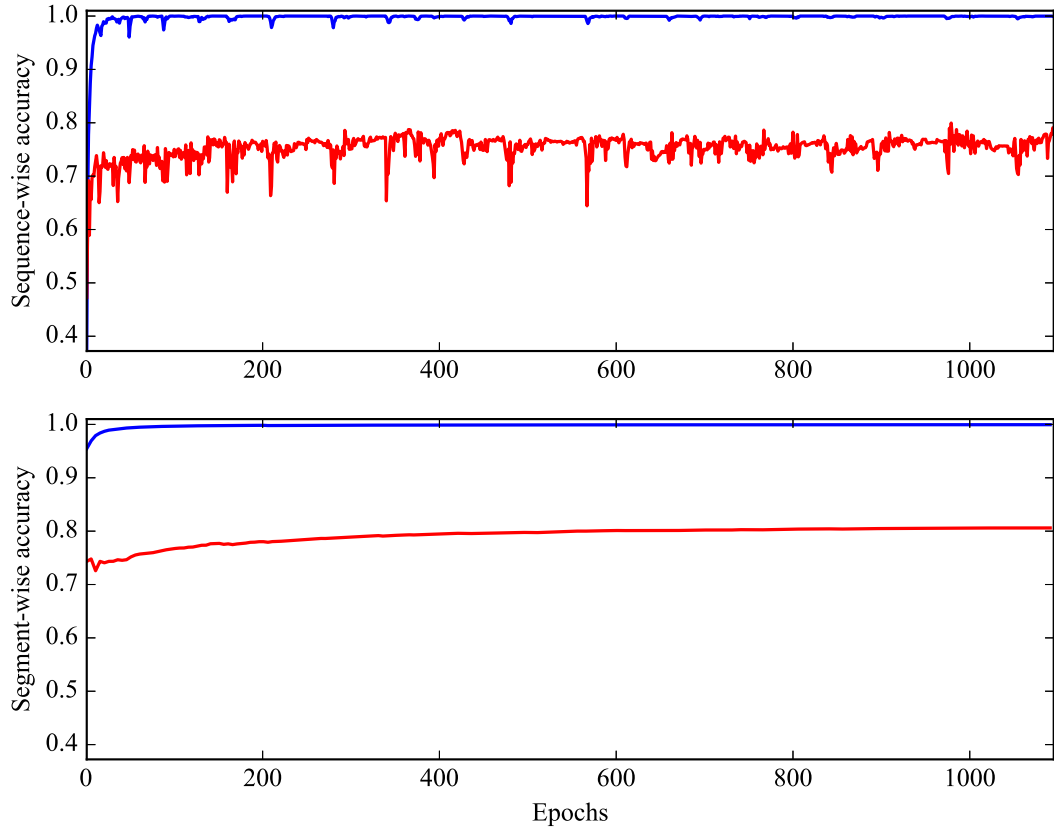


Figure 3.5: Saturation of segment and sequence-wise validation accuracies (in red) encountered during the non-full training phase.

training set, and this is why we call this phase full training. During full training we do not check the generalizing performance, but rather we assume, based on the non-full training phase, that its behaviour will saturate instead of reaching its highest value and then decrease. By doing so we can afford to increase the amount of training data from which the network will be able to learn more general representations. Different performances with full and non-full training setups are evaluated in Chapter 4.

Batch normalization This novel technique, introduced by Ioffe and Szegedy in 2015 [56], addresses a problem known as *internal covariate shift* [57]. Whenever weights of one layer are updated, the distribution of outputs of such layer will change consequentially. Due to this, also outputs’ statistics will vary over time. This phenomenon has an impact on weights of the next layer: they will have to adjust themselves in order to compensate such variations. This behaviour will affect in the same way also the next layers, therefore resulting in an overall slowed training.

Batch normalization (BN) is here introduced as an intermediate “layer” that will

take care of normalizing its input x by applying a linear transformation, indicated here with $\text{BN}_{\beta,\gamma}$ and given from:

$$y = \text{BN}_{\beta,\gamma}(x) = \frac{\gamma}{\sqrt{\text{Var}[x] + \epsilon}} \cdot x + \left(\beta - \frac{\gamma \text{E}[x]}{\sqrt{\text{Var}[x] + \epsilon}} \right), \quad (3.9)$$

where γ and β are the transformation parameters that will be learned during training, whether ϵ is a small value used in order not to divide numerators by zero. Moreover, $\text{E}[x]$ and $\text{Var}[x]$ are the mean and the variance of the input, calculated after a batch of input samples have propagated through the network.

As we can understand, BN is introduced in order to speed up the model training rather than properly regularize the model by counteracting overfitting. However, in [56, p. 6] the authors argue that “batch normalization provides similar regularization benefits as dropout, since the activations observed for a training example are affected by the random selection of examples in the same mini-batch”. Therefore we refer to it as a regularizing technique as well.

In our final configuration, a BN layer is used after each convolutional layer, normalizing each kernel output. This results in having one additional γ and β parameter for each kernel. Because of their number being very low ($\sim 10^2$), this will not drastically increment the overall number of parameters ($\sim 10^5$), therefore we believe that such complexity increment represents an advantageous trade-off with the expected performance enhancement. Comparisons between non-batch-normalized and batch-normalized networks are reported in Chapter 4.

4. EVALUATION

In this chapter we introduce the DCASE ASC development dataset and the metrics used for the evaluation of our system. Then we discuss experiments and comparisons performed between our model and other comparable architectures. Finally, a brief evaluation of our system with the DCASE 2013 database is reported.

4.1 Dataset and metrics

The DCASE 2016 ASC development database has been gathered by the Audio Research Group of the Tampere University of Technology in Finland between June 2015 and January 2016. The database contains 1170 different segments equally balanced between 15 different acoustic scenes, or classes. These predefined classes are: “beach”, “bus”, “café/restaurant”, “car”, “city center”, “forest path”, “grocery store”, “home”, “library”, “metro station”, “office”, “park”, “residential area”, “train” and “tram”. For each class 78 thirty-second segments are provided, totaling 39 minutes of audio per class. For evaluating the system, a four-fold cross-validation is performed. This means that the dataset splitting between training and test sets is performed four times, each time selecting different segments for the two subsets. By doing so we will obtain four networks with the same architecture, but different parameters, due to their dependence on the training data. Cross-validation leads to a considerable computation time increment, but it is a very common test used to evaluate the flexibility of the system when different training data is used. When all models are trained, an overall performance score is calculated between them, showing a more significant result than the single score calculated for only one model. The different training/test subsets (Table 4.1) are provided by the baseline system — see [58] for further details —, meaning that preliminary results between the challenge

Table 4.1: Training and test segments for each fold.

<i>fold</i>	<i>training segments</i>	<i>test segments</i>
1	880	290
2	880	290
3	872	298
4	878	292

participants can be compared.

In the dataset we can distinguish several groups of segments, each of which being a splitting of a single longer recording made in a specific location. Most of the classes show 12 different locations, while only “home” and “office” show 11 and ten locations respectively. Due to this, we observe that segments obtained from the same recording show very similar background noise and acoustic characteristics, hence only introducing new locations to the training set would result in having additive information. On the contrary, we do not consider the introduction of new segments belonging to an “already seen” location to be equally informative. This fact has a remarkable impact on the actual dimension of the dataset: in Section 4.4 we observe how the addition of new locations to the training set considerably improves the performance of many systems evaluated on the test set.

The system is designed to output an *overall score* which is the result of the calculation of subsequent segment-wise accuracies. First of all we calculate an ensemble of *fold-wise class accuracies*, their number being equal to the number of classes. To calculate each of these we must compare how many segments of a particular class have been detected by our system with respect to the annotated total, i.e. the ground truth. If we consider c and f to be the current class and fold respectively, the per-class accuracy (CA) is calculated as:

$$\text{CA}_c^{(f)} = \frac{r_c^{(f)}}{t_c^{(f)}}, \quad (4.1)$$

where $r_c^{(f)}$ is the number of correctly classified segments and $t_c^{(f)}$ the known total number of segments belonging to class c . Then, when all per-class accuracies are calculated, the fold accuracy (FA) is obtained as:

$$\text{FA}^{(f)} = \frac{1}{C} \sum_{c=1}^C \text{CA}_c^{(f)}, \quad (4.2)$$

therefore being an average between per-class accuracies. Finally, the overall accuracy (OA) is calculated by averaging the four per-fold accuracies:

$$\text{OA} = \frac{1}{F} \sum_{f=1}^F \text{FA}^{(f)}, \quad (4.3)$$

where F is the number of folds used for the cross-validation setup, that is four in our case.

4.1.1 Perceptual observations

During our experiments we performed some informal listening of segments. This was done in order to test our ability to classify segments belonging to frequently misclassified classes. Thanks to this listening we can notice that very similar background noises, coupled with non-characteristic sporadic sounds, can make the classification for some segments very complicated. For example, classes like “residential area” and “forest path” commonly show wind as background noise which, together with sounds like birds chirping, do not allow us to perform a completely certain classification. Moreover, long periods of silence are encountered in segments belonging to quiet scenes like “residential area”, “forest path”, “park”, and “home”, therefore making the classification task even harder. Misclassification can also rise from different interpretations given to a specific location. We can take as example one recording done in a restaurant wagon of a train. There, because of the presence of sounds like dishes and cutlery ticking, coupled with people talking more loudly than on ordinary wagons, it resulted difficult for us to correctly label this location as “train” rather than “restaurant”.

In conclusion, from our listening we can confirm that segments coming from the same location show very similar acoustic characteristics. Because of this, there are ensembles of segments that only represent a specific location in a particular hour of the day under fixed weather conditions, therefore not introducing significant information. On the contrary, by listening to segments coming from different locations it is possible to gather more information about the main characteristics of a particular class. For example, the “park” is mostly characterized by locations with a very quiet background. Probably because of this, a longer recording has been taken near a water flow, therefore introducing new information about a plausible acoustic park environment.

4.2 Baseline system

The baseline system is provided by the Tampere University of Technology and consists of a GMM-based classifier [58] operating with MFCC features. To obtain these coefficients the same pre-processing chain described in Chapter 2 is applied. At first, a 2048-points STFT, a 40 mel-bands filtering and a logarithmic conversion is performed on each frame, obtaining 40 log-mel coefficients. These coefficients are then transformed by means of a 20-points discrete cosine transform (DCT), finally obtaining the MFCCs. From them, 40 additional coefficients are calculated, i.e. delta and delta-delta coefficients. These latter coefficients are used to describe the dynamic of the audio evolution over time, therefore characterizing each frame with information derived from neighbouring frames, i.e. its context. This results in an

overall feature vector of 60 coefficients for each frame, which is used to feed the GMM classifier.

The baseline systems trains 15 different GMMs, each of which modeling a particular class and composed of 16 Gaussians. The training is performed with the expectation-maximization (EM) algorithm [59], which estimates, for each Gaussian, its mixture weight, mean and covariance. The training is done on all the training subset for 40 iterations of the EM algorithm, therefore no validation data is used. When the 15 mixtures are estimated, the system is able to calculate 15 log-probabilities (one per class) for all frames under each model. The segment classification is then performed by summing all frame-wise log-probabilities and picking the class corresponding to the highest of these sums.

4.3 Network parameter experiments

In this section we report some of the experiments and results that led us to choose the system configuration proposed in Chapter 3.

4.3.1 Kernel number

The first group of experiments reported here is a search for a good kernel configuration for the proposed model. For this search we train different networks on three-second sequences and non-full training setup. Then, we compare their training times and validation accuracy scores in order to find the best. We decide a-priori to double the number of kernels in the second layer with respect to the first, hence proposed configurations are shown as *first layer/second layer* kernel numbers. This increase is done in order to allow the network to build more high-level representations in the second layer with respect to the first one. Other parameters, such as RF areas and the optimizer configuration, are the same as described in Sections 3.2 and 3.3.

In order to find a good trade-off between a good representation capacity — i.e.

Table 4.2: Comparison between kernel numbers. Validation scores are averages of the best validation accuracies obtained during the training of each fold.

<i>kernels (first/second layer)</i>	<i>BN</i>	<i>epoch time (s)</i>	<i>validation score (%)</i>
64/128	no	6	67.4
64/128	yes	19	72.6
128/256	no	41	69.9
128/256	yes	50	73.1
256/512	no	56	71.2
256/512	yes	100	74.2

number of parameters — and sensible training time, we train the proposed network with and without BN. We decide to use BN instead of dropout due to its higher impact on the epoch time, thus allowing us to compare the “worst” scenarios in terms of training time. We highlight that training of the networks is performed on CSC — IT Center for Science’s cluster with NVIDIA Tesla K80 graphics processing units (GPUs). In addition, all epoch times shown hereafter are calculated by averaging the maximum and the minimum of the epoch time reported by Keras during the network training.

In these experiments we decide to use the non-full training setup so to monitor the validation performance during training. As Table 4.2 highlights, the use of BN always causes an increment of each epoch time, the smallest being nine additional seconds for the 128/256 configuration. This increment strictly depends on the hardware used for training, nonetheless this information is considered relevant in order to optimize training times of following experiments.

By looking at validation scores reported in Table 4.2 we can notice that the best score (74.1%) is reached by the 256/512 configuration, but at a high price in terms of epoch time: 100 seconds per epoch. Nonetheless, in view of the full training phase, we believe that a high capacity will make the best of the introduction of new training data. Because of this, we choose the 128/256 kernel configuration as the best trade-off between an affordable epoch time and a good representation capacity.

4.3.2 Regularization methods

Results we report here aim to explore differences between two widely-known methods used to regularize NNs, i.e. dropout and BM, respectively introduced in Chapter 2 and Chapter 3. Because of the different nature of these two methods, the following comparison is not only based on the overall performance on the validation set. Yet, also different behaviours in terms of training accuracies and convergence times are reported and highlighted.

In Table 4.3 we show how different techniques affect the convergence time (averaged for the four folds) and scores on training and validation subsets. For “low”

Table 4.3: Regularization techniques comparison.

<i>regularizer</i>	<i>avg. conv. time (epochs)</i>	<i>train score (%)</i>	<i>validation score (%)</i>
none	290	100.0	75.0
low dropout	150	98.0	73.0
high dropout	290	92.0	61.0
BN	170	100.0	76.5

and “high” dropout configurations we refer to dropout rates of $(0.0, 0.1, 0.1)$ and $(0.1, 0.25, 0.25)$ respectively, each of these numbers being the rate of dropped connections outgoing from the *input*, *first*, and *second* layer. Since convolutional layers already show lower number of parameters with respect to fully-connected layers, too high dropout rates might prevent kernels from learning sensible features. Because of this, no higher dropout rates have been tested. As Table 4.3 shows, the use of dropout results in faster convergence times and lower training scores. Since the gap between the training and the validation score is reducing as the dropout rate is increased, we can say that the system achieves lower overfitting. However, also a performance drop on the test set is encountered, which is not to be desired. More in detail, the score on the test set goes from 75.2% of the vanilla network to 67.9% of the network with high dropout. In addition, the low dropout network reaches a 70.5% score.

If we consider the use of BN we can see that the average convergence time (compared to the vanilla network) is almost 44% shorter. For a fair comparison we can consider the absolute epoch time (seconds) reported in Table 4.2. With each epoch taking nine seconds more (see rows three and four of Table 4.2), this percentage decreases to 32%. However, we can expect that better BN implementations could narrow this gap in the near future. Considering the absolute convergence time, this acceleration is not to be underestimated: it consists of more than one training hour less for each model. If we take into account the overall performance on the test set, very similar scores are achieved by the normalized and the non-normalized models, respectively 75.2% and 75.9%.

Results reported here prove that, for our system, BN is an advantageous addition to the proposed model. In addition, despite dropout manages to reduce the gap between training and validation scores, we observed that it also gives an overall score worsening on both validation and test sets.

4.4 Main experiments and results

In this section we report the main experiments performed with the proposed CNN, discussing the obtained results. In the first place we report tests aimed to evaluate the proposed CNN with different sequence length and training configurations. Then, a comparison between the proposed model and different classifiers is reported.

4.4.1 Sequence lengths and full-training

Recent studies [41] proved that the human ability to distinguish different acoustic scenes is based on the recognition of their most prominent sounds. In addition, the more time we can listen to the scene, the more this capability improves, due to the

Table 4.4: Accuracy comparison for the proposed model trained with different sequence lengths and training modes. Standard deviations refer to full training accuracies obtained for different weight initializations.

<i>sequence length (s)</i>	<i>accuracies (%)</i>		
	<i>non-full training</i>	<i>full training</i>	\pm <i>s.d.</i>
0.5	68.2	75.4	0.8
1.5	74.0	78.4	1.2
3	75.9	79.0	0.7
5	74.1	78.3	0.9
10	71.5	77.3	0.9
30	74.0	75.6	0.4

chance that new sounds are encountered. Because of this, we here aim to compare the use of different sequence lengths in order to see if the proposed model manages to take advantage from longer sequences as a human would. For our experiments we use very short (half and one and a half seconds), medium (three and five seconds) and long sequences (ten and 30 seconds). Moreover, as we described in Chapter 3, we compare the use of early stopping based on a validation accuracy check and an early stopping done after a fixed period, but with more training data. In Chapter 3 we called these two phases non-full and full training.

First of all, we evaluate the average convergence time with the non-full training setup for all sequence lengths. As shown in Table 4.3 the average convergence time is 170 epochs, with insignificant differences encountered for different sequence lengths. Beside convergence times, we also report overall average scores achieved on the test set. After this phase, full training without the validation set is performed for a fixed number of epochs. For the full training setup we performed multiple experiments with different network’s weight initialization, therefore we are able to report in Table 4.4 standard deviations for different full training experiments. The highest convergence time (encountered in the fourth fold) is 390 epochs, but this long period results in a very small improvement ($< 0.2\%$) of the validation score, due to the saturating behaviour reported in Fig. 3.5. Hence we decide to fix the number of epochs to 200, after which the training will be forced to stop. Because of the saturating behaviour showed in the figure we think that this period will allow the network to learn a sufficient amount of information in all four folds.

As we can see from Table 4.4, the use of very short or very long sequences negatively affects the system performance, so that the use of medium-length sequences appears to be the best choice. This conjecture is supported by scores reported for both full and non-full training configurations. However, the full training setup has

True label	beach	59	0	0	2	1	1	0	0	0	0	0	2	13	0	0
	bus	0	60	0	2	0	0	0	0	0	0	0	0	10	6	
	cafe/rest.	0	0	58	0	0	0	1	1	0	8	5	0	0	0	5
	car	0	0	0	71	0	0	0	0	0	0	0	0	1	5	1
	city cent.	0	0	0	0	73	0	0	0	0	1	0	0	4	0	0
	forest path	0	0	0	0	0	75	0	1	0	0	0	0	2	0	0
	groc. store	1	0	1	0	1	0	69	0	3	3	0	0	0	0	0
	home	0	0	1	0	0	2	0	63	3	0	6	2	1	0	0
	library	0	1	1	0	0	0	5	10	52	2	1	0	1	1	4
	metro st.	0	0	0	0	0	0	1	1	0	75	0	0	0	0	1
	office	0	0	0	0	0	1	0	1	0	0	76	0	0	0	0
	park	0	0	0	0	0	1	0	0	1	0	3	46	27	0	0
	resid. area	1	0	0	0	2	3	0	0	0	1	0	14	57	0	0
	train	0	23	7	2	0	0	0	0	0	0	0	0	1	36	9
	tram	0	2	0	2	1	0	0	0	0	1	0	0	5	6	61
		Predicted label														

Figure 4.1: Confusion matrix for the fully-trained CNN. Each value is the number of classified segments among all the four folds. Three-second sequences are used.

been tested for only four different random weights initializations, hence more tests should be performed in order to confirm or reject these differences with statistical significance. Conversely, the full training setup always proves to give a significant performance improvement for all sequence configurations, reaching a peak of 79.0% with the three-second configuration.

In Fig. 4.1 we show the confusion matrix obtained on the test set with the proposed CNN operating on three-second sequences. The matrix shows an appreciable diagonal, meaning that many classes are correctly classified. However, some classes are strongly or even drastically misclassified: we refer to the class-wise accuracies obtained with “train” (46%) and “park” (59%). The reported confusion matrix mostly confirms what has been observed in Section 4.1.1: differences between some of the mostly misclassified classes resulted difficult to distinguish also during our listening.

4.4.2 Different model comparisons

The best-performing network is here compared to other models which are designed to have representation capacities comparable to our system’s. The first model used

for the comparison is a two-layer fully-connected neural network (or MLP). Since the input to this network is connected to all neurons in the first hidden layer, the use of long sequences has been considered computationally too expensive. The classification is therefore performed frame-wise, with the input being the spectrogram of a context window of 15 frames and the output the class associated with the central frame. The context window is slid frame-by-frame among the whole segment. At last, the class associated with the segment is calculated as described in Eq. (3.3), where N is now the number of frames in the segment.

The second network tested during our experiments closely resembles the CNN proposed by Phan et al. in [60] for an audio event recognition task. Based on it, we test this different CNN whose inputs are the same sequences used for the proposed model. This CNN shows only one convolutional layer whose kernel's RFs cover all the log-mel bands and a time context window of 15 frames, i.e. 15×60 area. Feature maps outputted from this layer are then arrays, since the frequency dimension results in being shrank by the convolutional kernels. A max-pooling layer is then used to compress also the time dimension, hence giving us only one scalar as output of the sub-sampling layer. Finally, a softmax layer is used to output the classification scores for the input sequence. The class for the whole segment is always obtained as defined in Eq. (3.3).

In Table 4.5 we report all accuracy scores obtained for all described models, including the proposed one. In addition, we compare our system to the baseline when both are fed with MFCC features.

From Table 4.5 we can see that the MLP results in being the worst performing architecture, with 69.3% score for a full training setup. The same architecture, trained with the non-full training setup, achieved 66.6% overall score. By looking at the second row of Table 4.5, we can see that the described one-layer CNN succeed to improve the MLP score by a significant 5.5% and 3.7% after a full and a non-full training respectively. However, the proposed model manages to improve scores obtained by all other neural architectures by achieving an overall 79.0% score with

Table 4.5: Accuracy comparison for different systems and training modes ("non-full" and "full" training).

<i>system</i>	<i>sequence length (s)</i>	<i>accuracies (%)</i>	
		<i>non-full</i>	<i>full</i>
two-layer MLP (log-mel)	-	66.6	69.3
one-layer CNN (log-mel)	3	70.3	74.8
two-layer CNN (log-mel)	3	75.9	79.0
two-layer CNN (MFCC)	5	67.7	72.6
baseline GMM (MFCC)	-	-	72.6

a full-training. It is interesting to notice that this score drops to 72.6% if MFCCs are used, meaning that, with these features, the performance equals the baseline system's.

4.5 DCASE evaluations

In this section we briefly report the evaluation of the proposed system on the DCASE 2013 and 2016 evaluation datasets. In both cases we use the same architecture and features described in Chapter 3, with a three-second sequence configuration. Moreover, a comparison between our system and the best-performing competitor systems is shown.

4.5.1 DCASE 2016

The DCASE 2016 evaluation dataset consists of 390 30-second audio segments equally divided into 15 classes (already listed in Section 4.1). These segments are obtained similarly to those provided for the development dataset: long recordings were done in new locations and then split into 30-second audio segments.

For this evaluation no cross-validation is used, therefore we train only one model on the whole development dataset (1170 segments). As first step, a non-full training is performed by keeping two locations (and so their respective segments) for the validation check. This phase proved that the saturating behaviour of the segment-wise validation accuracy is once more maintained. Due to this we can perform a full training for the estimated 400 epochs and obtain the final model which we use to calculate predictions on the evaluation dataset. Due to the large amount of submissions for the DCASE 2016 ASC task, in Table 4.6 we report only the seven best performing ones.

The best accuracy score is obtained by a fusion between two different systems [61] reaching 89.6% accuracy on the evaluation dataset. The former system is based on the extraction from MFCC features of a novel, higher-level, feature representation,

Table 4.6: Accuracy comparison for the seven best systems (out of 49) tested on the DCASE 2016 evaluation dataset.

<i>features</i>	<i>system</i>	<i>accuracies (%)</i>	<i>reference</i>
MFCC + spectrogram	fusion	89.7	[61]
MFCC	i-vector	88.7	[61]
spectrogram	NMF	87.7	[62]
various features	fusion	87.2	[63]
MFCC	i-vector	86.4	[61]
various features	fusion	86.4	[64, 65]
log-mel	proposed CNN	86.2	-

known as i-vector [66], whereas the latter system is a deep CNN trained on audio spectrograms. Output of the two systems are then combined to obtain the final predicted classes. Other i-vector experiments have been conducted by the same group, therefore in Table 4.6 we can notice that this system alone manages to reach two more very good scores (88.7% and 86.4%) on the evaluation dataset.

Another good accuracy score (87.7%) is reached by a system based on non-negative matrix factorization (NMF) [62]. This technique, introduced in [67], is based on the learning of a dictionary of atoms whose weighted combinations are able to exhaustively represent the training data. In their work authors propose a slight modification to the usual learning algorithm that proves to give better results on the DCASE 2016 development dataset.

In [63] authors make use of five different sets of features to train an ensemble of three systems: two GMM models and a nearest neighbour classifier. Outputs of the three systems are then normalized and fused together to obtain the final classification. This system ranked the fourth place with 87.2% accuracy score.

Finally, another fusion system (described in [64, 65]) is designed to combine outputs from a MLP trained on spectral features with the “one-against-all” paradigm and a nearest neighbour classifier. This latter classifier is trained on an ensemble of different features (spectral, cepstral, and voicing-related) represented by multi-scale Gaussian kernels in a low-dimensional feature space.

4.5.2 DCASE 2013

As mentioned in Section 2.4, this dataset consists of 100 30-second audio segments equally divided into ten classes: “bus”, “busy street”, “office”, “open air market”, “park”, “quiet street”, “restaurant”, “supermarket”, “tube”, and “tube station”. The system evaluation is now based on a five-fold cross-validation, each fold having 80 segments to be used for training and 20 for testing. It is evident that the amount

Table 4.7: Accuracy comparison for different systems tested on the DCASE 2013 evaluation dataset. Means and standard deviations are calculated on accuracies obtained in the five folds.

<i>features</i>	<i>system</i>	<i>accuracies (%)</i>	<i>± s.d.</i>	<i>reference</i>
MFCC	baseline	55.0	10.0	[50]
various features	HMM - GMM	65.0	6.9	[51]
various features	SVM	69.0	12.0	[52]
wavelet and MFCC	tree bagger	72.0	8.4	[68]
RQA ^(*) features	SVM	76.0	7.2	[53]
log-mel	proposed CNN	77.0	2.7	-

(*): recurrent quantification analysis.

of data available for this challenge is more limited if compared to the DCASE 2016 dataset, but nonetheless we now have that each segment is a recording coming from a different location. This means that each class is represented by ten segments precisely corresponding to ten locations.

The evaluation we report here aims to test the capability of our system and training method to adapt to new datasets without being fine-tuned through preliminary tests. In each fold we firstly train our model by monitoring its generalizing performance on a validation subset of 20 files, thus keeping only 60 files for the effective training. Similar saturating behaviours are encountered for the segment-wise validation accuracy in all folds, with the exception of one fold in which the score keeps slowly increasing until the training is forced to stop at the 2500th epoch. Mainly because of this, the estimated convergence time on the non-full training setup is 500 epochs, thus this is the epoch time used for the following full training phase. In Table 4.7 we show a comparison between our system trained with full training setup (last row) and the best performing systems presented for the DCASE 2013 challenge, already introduced at the end of Chapter 2.

4.6 Discussion

Results reported in the first part of Section 4.4 highlight different aspects of the proposed model. In the first place, we can notice from Table 4.4 that, differently from what we expected, the use of long sequences (equal or more than ten seconds) does not help the model to improve the classification performance. This can result from a strong reliance of the network only on the scene’s background noise. Intuitively, a short period is already enough to have good background information. Because of this, very ambiguous environments could be better classified when the presence of a difficult sequence is muffled by its combination with other sequences, perhaps easier to recognize. On the contrary, the use of the whole segment (30-second sequence) does not guarantee that the network will become able to recognize more significant acoustic characteristics, e.g. the presence or repetition of particular sounds. Therefore, if the background noise is difficult to distinguish from the others, the combination of fewer sequences can mislead the correct classification of the whole segment.

Table 4.4 shows how the addition of new informative training data — i.e. new locations — results in significant performance improvements (from +1.6% to +7.2%) for all sequence lengths. The importance of having new locations among the training data is comprehensible if we look at each segment as a point in an hypothetical “information space”. In this space, we can visualize segments coming from the same location as points grouped in a specific restricted area. Due to this, a good covering of this space can be achieved only by adding a new segment recorded in a completely

different location. On the contrary, segments coming from the same location do not bring much new or relevant information. This problem can only be solved with the introduction of new locations (i.e. gathering more data) or by using one or more data augmentation techniques. We believe that this latter solution can be an interesting area of research since, according to our knowledge, no data augmentation technique has ever been specifically designed for ASC.

In conclusion, it is interesting to notice that the comparison between the use of MFCC for our model and the baseline system ends in a fair result. This suggests that, for our model and task, the use of MFCCs is not to be preferred to log-mel energies. The main difference between these two features lies in the higher incorrelation introduced by the DCT for MFCCs with respect to log-mel energies. Hence, this result suggests that the use of more correlated features helps the network in building more significant representations, probably due to the intrinsic information carried by the correlation itself.

5. CONCLUSIONS

In this work we showed how a CNN can be implemented for solving the task of acoustic scene classification. The data representation we use is the log-mel spectrogram which, given its little need for pre-processing, is considered a low-level feature representation. The use of this feature for a convolutional architecture is supported by the capability of these models to autonomously learn underlying information from their training data, hence projecting the input in higher-level feature spaces.

With a first series of preliminary experiments we showed how a recently introduced regularization technique –i.e. batch normalization— can be used in order to sensibly speed up the network training and also to improve its generalizing performance. This technique proved to give better results if compared to dropout, another widespread regularization technique.

The proposed model has been tested with different spectrogram time lengths in order to discover if using longer contexts would benefit the network’s recognition ability. In addition, two training phases are introduced: non-full and full training. During non-full training we monitor the generalization performance on a validation set and also increase the sequence variability by permuting and time-shifting training spectrograms. The saturation of the validation accuracy allowed us to give up the validation check and re-train the model with the training set in its entirety. By doing so, we obtain another significant performance boost despite the intrinsic difficulty in finding the best moment in which to stop the parameter update.

We proved that, with the described setup, our system performs best when working with three-second sequence length. We believe that this is due to a propensity to better learn the sequence background noise rather than single acoustic event occurrences. Due to this, our system reaches very high class accuracies with non-ambiguous classes, like the forest path (96.2%) and the metro station (96.2%). On the contrary, we noticed that very similar classes are highly misclassified. The most prominent examples of this are the park and train classes which are mostly recognized as residential area (34.6% of the time) and bus (29.5% of the time) respectively.

By some comparisons performed with other neural architectures, we proved that the proposed system performs better than a two-layer MLP and a similar, but shallower, CNN with more kernels. Both networks were designed in order to have

similar amounts of parameters with respect to the proposed model's. Moreover, we showed that our model managed to outperform more than 40 systems submitted to the DCASE 2016 ASC challenge, ranking the sixth place with 86.2% accuracy score. This result is the highest among all accuracies obtained with a pure CNN system, also outranking a CNN ensemble.

Finally, we tested the proposed architecture on the unknown DCASE 2013 challenge dataset. In order to evaluate the system flexibility, no fine-tuning has been done before this last test. On this dataset the average score obtained on five different folds is 77%, which consists of a slight (1%) improvement with respect to the best performing system.

We believe that future investigation should address the developing of augmentation techniques specifically designed for ASC. Such algorithms, aimed at increasing the available training data, would allow researchers to use deeper networks without the need of collecting and labelling more recordings. Furthermore, new low-level features could be added to the log-mel spectrogram in order to enhance the network performance. In this scenario it would be necessary to investigate how to represent them — e.g. giving them the “spatiality” needed for convolutional layers — and how to couple them with spectrograms; for example it could be possible to feed them to another network and then merge the two NNs.

REFERENCES

- [1] P. Simon. “Too Big to Ignore: The Business Case for Big Data”, vol. 72, pp. 89. John Wiley & Sons. 2013.
- [2] R. F. Lyon. “Machine Hearing: An Emerging Field”. *IEEE Signal Processing Magazine*, vol. 27, no. 5, pp. 131–139. 2010.
- [3] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. “Going Deeper with Convolutions”. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1–9. 2015.
- [4] A. Mordvintsev, C. Olah, and M. Tyka. “Inceptionism: Going Deeper into Neural Networks” Accessed in: 2015. URL: <https://research.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html>.
- [5] A. Øigard. “Visualizing GoogLeNet Classes” Accessed in: 2015. URL: <http://auduno.com/post/125362849838/visualizing-googlenet-classes>.
- [6] G. McCaig, S. DiPaola, and L. Gabora. “Deep Convolutional Networks as Models of Generalization and Blending Within Visual Creativity”. 2016.
- [7] B. Schilit, N. Adams, and R. Want. “Context-aware Computing Applications”. *Workshop on Mobile Computing Systems and Applications*, pp. 85–90. 1994.
- [8] Y. Xu, W. J. Li, and K. K. Lee. “Intelligent Wearable Interfaces”. John Wiley & Sons. 2008.
- [9] S. Chu, S. Narayanan, C.-C. J. Kuo, and M. J. Mataric. “Where am I? Scene Recognition for Mobile Robots using Audio Features”. *International Conference on Multimedia and Expo*, pp. 885–888. 2006.
- [10] E. Cakir, T. Heittola, H. Huttunen, and T. Virtanen. “Polyphonic Sound Event Detection using Multi Label Deep Neural Networks”. *International Joint Conference on Neural Networks (IJCNN)*, pp. 1–7. 2015.
- [11] G. Parascandolo, H. Huttunen, and T. Virtanen. “Recurrent Neural Networks for Polyphonic Sound Event Detection in Real Life Recordings”. *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 6440–6444. 2016.
- [12] A. Graves, A.-r. Mohamed, and G. Hinton. “Speech Recognition with Deep Recurrent Neural Networks”. *International Conference on Acoustics, Speech and Signal Processing*, pp. 6645–6649. 2013.

- [13] S. Böck and M. Schedl. “Polyphonic Piano Note Transcription with Recurrent Neural Networks”. *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 121–124. 2012.
- [14] C. C. Paige and M. A. Saunders. “LSQR: An Algorithm for Sparse Linear Equations and Sparse Least Squares”. *Transactions on Mathematical Software (TOMS)*, vol. 8, no. 1, pp. 43–71. 1982.
- [15] Y. N. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, and Y. Bengio. “Identifying and Attacking the Saddle Point Problem in High-dimensional Non-convex Optimization”. *Advances in Neural Information Processing Systems*, pp. 2933–2941. 2014.
- [16] A. Choromanska, M. Henaff, M. Mathieu, G. B. Arous, and Y. LeCun. “The Loss Surfaces of Multilayer Networks”. 2014.
- [17] S. Lawrence and C. L. Giles. “Overfitting and Neural Networks: Conjugate Gradient and Backpropagation”. *Proceedings of the IEEE-INNS-ENNS International Joint Conference*, pp. 114–119. 2000.
- [18] F. Rosenblatt. “The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain”. *Psychological Review*, vol. 65, no. 6, pp. 386. 1958.
- [19] T. M. Mitchell. “Machine Learning”, pp. 88–95. McGraw-Hill. 1997.
- [20] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. “Efficient Backprop”. *Neural Networks: Tricks of the Trade*, pp. 9–48. 2012.
- [21] X. Glorot, A. Bordes, and Y. Bengio. “Deep Sparse Rectifier Neural Networks”. *International Conference on Artificial Intelligence and Statistics (AISTATS)*, no. 106, pp. 315–323. 2011.
- [22] S. Grossberg. “Contour Enhancement, Short Term Memory, and Constancies in Reverberating Neural Networks”. Springer. 1973.
- [23] K. Hornik, M. Stinchcombe, and H. White. “Multilayer Feedforward Networks are Universal Approximators”. *Neural Networks*, vol. 2, no. 5, pp. 359–366. 1989.
- [24] P. A. Devijver and J. Kittler. “Pattern Recognition: A Statistical Approach”. Prentice Hall. 1982.
- [25] D. H. Hubel and T. N. Wiesel. “Receptive Fields and Functional Architecture of Monkey Striate Cortex”. *The Journal of Physiology*, vol. 195, no. 1, pp. 215–243. 1968.

- [26] A. Krizhevsky, I. Sutskever, and G. E. Hinton. “Imagenet Classification with Deep Convolutional Neural Networks”. *Advances in Neural Information Processing Systems*, pp. 1097–1105. 2012.
- [27] H. J. Kelley. “Gradient Theory of Optimal Flight Paths”. *Ars Journal*, vol. 30, no. 10, pp. 947–954. 1960.
- [28] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. “Learning Representations by Back-propagating Errors”. *Cognitive Modeling*, vol. 5, no. 3, pp. 1. 1988.
- [29] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. “On the Importance of Initialization and Momentum in Deep Learning”. *Proceedings of the 30th International Conference on Machine Learning*, pp. 1139–1147. 2013.
- [30] A. Y. Ng. “Feature selection, L1 vs. L2 Regularization, and Rotational Invariance”. *Proceedings of the Twenty-first International Conference on Machine learning*, pp. 78. 2004.
- [31] J. Kittler, M. Hatef, R. P. Duin, and J. Matas. “On Combining Classifiers”. *Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 3, pp. 226–239. 1998.
- [32] G. Rogova. “Combining the Results of Several Neural Network Classifiers”. *Neural Networks*, vol. 7, no. 5, pp. 777–781. 1994.
- [33] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958. 2014.
- [34] L. Hertel, H. Phan, and A. Mertins. “Comparing Time and Frequency Domain for Audio Event Recognition Using Deep Learning”. *arXiv preprint arXiv:1603.05824*. 2016.
- [35] D. J. Defatta, J. G. Lucas, and W. S. Hodgkiss. “Digital Signal Processing”. John Wiley & Sons, Inc. 1989.
- [36] S. S. Stevens, J. Volkman, and E. B. Newman. “A Scale for the Measurement of the Psychological Magnitude Pitch”. *The Journal of the Acoustical Society of America*, vol. 8, no. 3, pp. 185–190. 1937.
- [37] A. S. Bregman. “Auditory Scene Analysis: The Perceptual Organization of Sound”. MIT press. 1994.
- [38] D. Wang and G. J. Brown. “Computational Auditory Scene Analysis: Principles, Algorithms, and Applications”. Wiley-IEEE Press. 2006.
- [39] P. Divenyi. “Speech Separation by Humans and Machines”. Springer Science & Business Media. 2004.

- [40] M. Slaney. “The History and Future of CASA”. *Speech Separation by Humans and Machines*, pp. 199–211. 2005.
- [41] V. T. Peltonen, A. J. Eronen, M. P. Parviainen, and A. P. Klapuri. “Recognition of Everyday Auditory Scenes: Potentials, Latencies and Cues”. *Preprints of the Audio Engineering Society*. 2001.
- [42] N. Sawhney and P. Maes. “Situational Awareness from Environmental Sounds”. *MIT Media Lab Technical Report*. 1997.
- [43] B. Clarkson, N. Sawhney, and A. Pentland. “Auditory Context Awareness via Wearable Computing”. *Energy*, vol. 400, no. 600, pp. 20. 1998.
- [44] B. Clarkson and A. Pentland. “Extracting Context from Environmental Audio”. *Second International Symposium on Wearable Computers*, pp. 154–155. 1998.
- [45] Z. Liu, Y. Wang, and T. Chen. “Audio Feature Extraction and Analysis for Scene Segmentation and Classification”. *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology*, vol. 20, no. 1-2, pp. 61–79. 1998.
- [46] V. Peltonen, J. Tuomi, A. Klapuri, J. Huopaniemi, and T. Sorsa. “Computational Auditory Scene Recognition”. *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pp. II–1941. 2002.
- [47] K. El-Maleh, A. Samouelian, and P. Kabal. “Frame Level Noise Classification in Mobile Environments”. *International Conference on Acoustics, Speech, and Signal Processing*, pp. 237–240. 1999.
- [48] A. J. Eronen, V. T. Peltonen, J. T. Tuomi, A. P. Klapuri, S. Fagerlund, T. Sorsa, G. Lorho, and J. Huopaniemi. “Audio-based Context Recognition”. *Transactions on Audio, Speech, and Language Processing*, vol. 14, no. 1, pp. 321–329. 2006.
- [49] N. Sawhney. “Contextual Awareness, Messaging and Communication in Nomadic Audio Environments”. PhD thesis. Massachusetts Institute of Technology, 1998.
- [50] D. Giannoulis, E. Benetos, D. Stowell, M. Rossignol, M. Lagrange, and M. D. Plumbley. “Detection and Classification of Acoustic Scenes and Events: An IEEE AASP challenge”. *Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA)*, pp. 1–4. 2013.
- [51] M. Chum, A. Habshush, A. Rahman, and C. Sang. “IEEE AASP Scene Classification Challenge using Hidden Markov Models and Frame Based Classification”. *IEEE AASP Challenge on Detection and Classification of Acoustic Scenes and Events*. 2013.

- [52] J. T. Geiger, B. Schuller, and G. Rigoll. “Large-scale Audio Feature Extraction and SVM for Acoustic Scene Classification”. *Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA)*, pp. 1–4. 2013.
- [53] G. Roma, W. Nogueira, P. Herrera, and R. de Boronat. “Recurrence Quantification Analysis Features for Auditory Scene Classification”. *IEEE AASP Challenge: Detection and Classification of Acoustic Scenes and Events, Tech. Rep.* 2013.
- [54] B. McFee, C. Raffel, D. Liang, D. P. Ellis, M. McVicar, E. Battenberg, and O. Nieto. “Librosa: Audio and Music Signal Analysis in Python”. *Proceedings of the 14th Python in Science Conference*. 2015.
- [55] D. Kingma and J. Ba. “Adam: A Method for Stochastic Optimization”. *arXiv preprint arXiv: 1412.6980*. 2014.
- [56] S. Ioffe and C. Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. *arXiv preprint arXiv: 1502.03167*. 2015.
- [57] H. Shimodaira. “Improving Predictive Inference Under Covariate Shift by Weighting the Log-likelihood Function”. *Journal of Statistical Planning and Inference*, vol. 90, no. 2, pp. 227–244. 2000.
- [58] A. Mesaros, T. Heittola, and T. Virtanen. “TUT Database for Acoustic Scene Classification and Sound Event Detection”. *24th European Signal Processing Conference (EUSIPCO)*. 2016.
- [59] L. Xu and M. I. Jordan. “On Convergence Properties of the EM Algorithm for Gaussian Mixtures”. *Neural Computation*, vol. 8, no. 1, pp. 129–151. 1996.
- [60] H. Phan, L. Hertel, M. Maass, and A. Mertins. “Robust Audio Event Recognition with 1-Max Pooling Convolutional Neural Networks”. *arXiv preprint arXiv: 1604.06338*. 2016.
- [61] H. Eghbal-Zadeh, B. Lehner, M. Dorfer, and G. Widmer. “CP-JKU Submissions for DCASE-2016: A Hybrid Approach Using Binaural I-Vectors and Deep Convolutional Neural Networks”. *IEEE AASP Challenge on Detection and Classification of Acoustic Scenes and Events (DCASE)*. 2016.
- [62] V. Bisot, R. Serizel, S. Essid, and G. Richard. “Supervised Nonnegative Matrix Factorization for Acoustic Scene Classification”. *IEEE AASP Challenge on Detection and Classification of Acoustic Scenes and Events (DCASE)*. 2016.
- [63] S. Park, S. Mun, Y. Lee, and H. Ko. “Score Fusion of Classification Systems for Acoustic Scene Classification”. *IEEE AASP Challenge on Detection and Classification of Acoustic Scenes and Events (DCASE)*. 2016.

- [64] E. Marchi, D. Tonelli, X. Xu, F. Ringeval, J. Deng, S. Squartini, and B. Schuller. “Pairwise Decomposition with Deep Neural Networks and Multiscale Kernel Subspace Learning for Acoustic Scene Classification”. *24th Acoustic Scene Classification Workshop 2016 European Signal Processing Conference (EUSIPCO)*. 2016.
- [65] E. Marchi, D. Tonelli, X. Xu, F. Ringeval, J. Deng, and B. Schuller. “The Up System for The 2016 DCASE Challenge Using Deep Recurrent Neural Network and Multiscale Kernel Subspace Learning”. *IEEE AASP Challenge on Detection and Classification of Acoustic Scenes and Events (DCASE)*. 2016.
- [66] N. Dehak, P. J. Kenny, R. Dehak, P. Dumouchel, and P. Ouellet. “Front-end Factor Analysis for Speaker Verification”. *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 19, no. 4, pp. 788–798. 2011.
- [67] D. D. Lee and H. S. Seung. “Learning the Parts of Objects by Non-negative Matrix Factorization”. *Nature*, vol. 401, no. 6755, pp. 788–791. 1999.
- [68] D. Li, J. Tam, and D. Toub. “Auditory Scene Classification using Machine Learning Techniques”. *IEEE AASP Challenge on Detection and Classification of Acoustic Scenes and Events (DCASE)*. 2013.